# Http

附录 <https://www.cnblogs.com/rickiyang/p/13138574.html>

# Https

附录https://blog.csdn.net/JavaMrZhang/article/details/52936250

# Spring

### 什么是spring

1. 用java编写的一个轻量级的开源框架, 主要用于java 应用的开发
2. 目标是简化企业级开发
3. 核心技术是IOC和AOP
4. IOC 控制反转, 将java对象的控制权较给ioc容器进行管理, 实现了业务层之间的解耦合
5. Aop 面向切面编程(代理), 把应用的业务逻辑和系统业务分开, 典型的有事务管理, Dubugging调试等

### Spring 的核心模块

1. Spring core: 核心容器提供Spring框架的基本功能, Spring以bean的方式组织和管理Java应用中的各个组件及其关系. Spring使用BeanFactory来产生和管理Bean, 它是工厂模式的实现. BeanFactory使用控制反转(IoC)模式将应用的配置和依赖性规范与实际的应用程序代码分开
2. Spring context: Spring上下文是一个配置文件,向Spring框架提供上下文信息. Spring上下文包括企业服务, 如JNDI、EJB、电子邮件、国际化、校验和调度功能。
3. Spring aop: 通过配置管理特性, Spring AOP 模块直接将面向方面的编程功能集成到了Spring框架中. 所以, 可以很容易地使 Spring框架管理的任何对象支持 AOP. Spring AOP 模块为基于 Spring 的应用程序中的对象提供了事务管理服务。通过使用 Spring AOP, 不用依赖 EJB 组件(提供公共服务), 就可以将声明性事务管理集成到应用程序中。
4. Spring dao： JDBC、DAO的抽象层提供了有意义的异常层次结构，可用该结构来管理异常处理，和不同数据库供应商所抛出的错误信息。异常层次结构简化了错误处理，并且极大的降低了需要编写的代码数量，比如打开和关闭链接。
5. Spring orm: Spring框架插入了若干个ORM框架，从而提供了ORM对象的关系工具，其中包括了Hibernate、JDO和 IBatis SQL Map等，所有这些都遵从Spring的通用事物和DAO异常层次结构。
6. Spring web: Web上下文模块建立在应用程序上下文模块之上，为基于web的应用程序提供了上下文。所以Spring框架支持与Struts集成，web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。
7. Spring web mvc : MVC框架是一个全功能的构建Web应用程序的MVC实现。通过策略接口，MVC框架变成为高度可配置的。MVC容纳了大量视图技术，其中包括JSP、POI等，模型来有JavaBean来构成，存放于m当中，而视图是一个街口，负责实现模型，控制器表示逻辑代码，由c的事情。Spring框架的功能可以用在任何J2EE服务器当中，大多数功能也适用于不受管理的环境。Spring的核心要点就是支持不绑定到特定J2EE服务的可重用业务和数据的访问的对象，毫无疑问这样的对象可以在不同的J2EE环境，独立应用程序和测试环境之间重用。

#### EJB是什么

EJB 是过去使用的一种技术, 就是把业务逻辑从客户端中拿出来放到服务器中, 通过RMI进行远程通信.

附: <https://blog.csdn.net/jojo52013145/article/details/5783677>

#### RMI

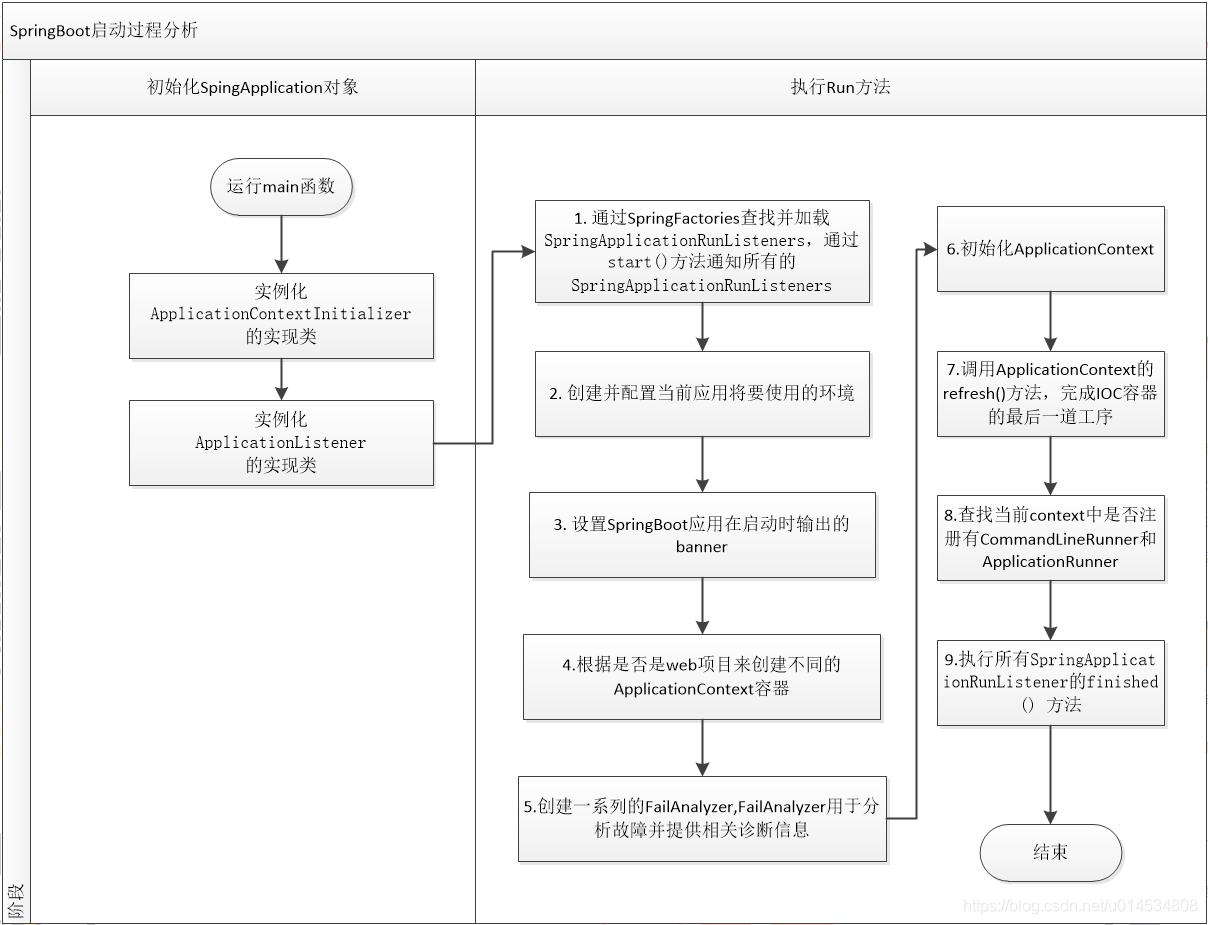
对象序列化: 通过id将java对象变成字节码, 可以存储通信恢复

RPC: remote process call 远程过程调用

RMI: RPC + 对象序列化 即: 远程类方法调用

# Spring Boot

## 启动过程



## 常用接口介绍

### Aware 接口

实现该接口的类必须是接受Spring 容器的管理才会生效, 项目运行时这类接口的方法会被逐一调用, 主要用来设置参数(取得环境上下文参数对象).

#### ImportAware

void setImportMetadata(AnnotationMetadata var1), 该方法主要是用来取得注解的属性信息的

#### BeanClassLoaderAware

void setBeanClassLoader(ClassLoader var1), 取得加载该Bean的类加载器

#### ApplicationContextAware

void setApplicationContext(ApplicationContext var1) 取得环境上下文

#### MessageSourceAware

void setMessageSource(MessageSource var1) 国际化配置信息的取得

#### ApplicationEventPublisherAware

void setApplicationEventPublisher(ApplicationEventPublisher var1) 发布器取得

### Bean 的回调接口

#### InitializingBean

void afterPropertiesSet() Bean 实例化后调用的回调方法(2).

#### DisposableBean

void destroy() Bean 销毁前回调该接口(2).

#### SmartInitializingSingleton

void afterSingletonsInstantiated() 所有的单例Bean 加载完成后回调该接口.

### 内置事件

**ContextRefreshedEvent**

ApplicationContext 被初始化或刷新时，该事件被发布。这也可以在 ConfigurableApplicationContext接口中使用 refresh() 方法来发生。此处的初始化是指：所有的Bean被成功装载，后处理Bean被检测并激活，所有Singleton Bean 被预实例化，ApplicationContext容器已就绪可用

**ContextStartedEvent**

当使用 ConfigurableApplicationContext （ApplicationContext子接口）接口中的 start() 方法启动 ApplicationContext 时，该事件被发布。你可以调查你的数据库，或者你可以在接受到这个事件后重启任何停止的应用程序。

**ContextStoppedEvent**

当使用 ConfigurableApplicationContext 接口中的 stop() 停止 ApplicationContext 时，发布这个事件。你可以在接受到这个事件后做必要的清理的工作。

**ContextClosedEvent**

当使用 ConfigurableApplicationContext 接口中的 close() 方法关闭 ApplicationContext 时，该事件被发布。一个已关闭的上下文到达生命周期末端；它不能被刷新或重启。

**RequestHandledEvent**

这是一个 web-specific 事件，告诉所有 bean HTTP 请求已经被服务。只能应用于使用DispatcherServlet的Web应用。在使用Spring作为前端的MVC控制器时，当Spring处理用户请求结束后，系统会自动触发该事件。

### 其它接口

#### Beanfactorypostprocesser

用于动态的注入bean

#### AutowireCapableBeanFactory

用于装载非IOC容器之中的Bean的, 通常用于第三方框架.

#### ObjectProvider

通常用于构造函数多参情况下, 隐式注入, 由编码方式进行延迟调用。

### 常用注解

##### @Import

1. 引入其他的@Configuration
2. 直接初始化其他类使之成为Bean
3. 指定实现ImportSelector(以及DefferredServiceImportSelector)的类, 用于个性化加载
4. 指定实现ImportBeanDefinitionRegistrar的类, 用于个性化加载

##### @DependsOn

用于控制Bean的加载顺序.

##### @ConfigurationProperties

注解一个类, 用配置文件中的值进行属性的赋值.

##### @EnableAutoConfiguration

该注解用于使得@ConfigurationProperties注解的类生效, 成为一个Bean.

### 参数校验

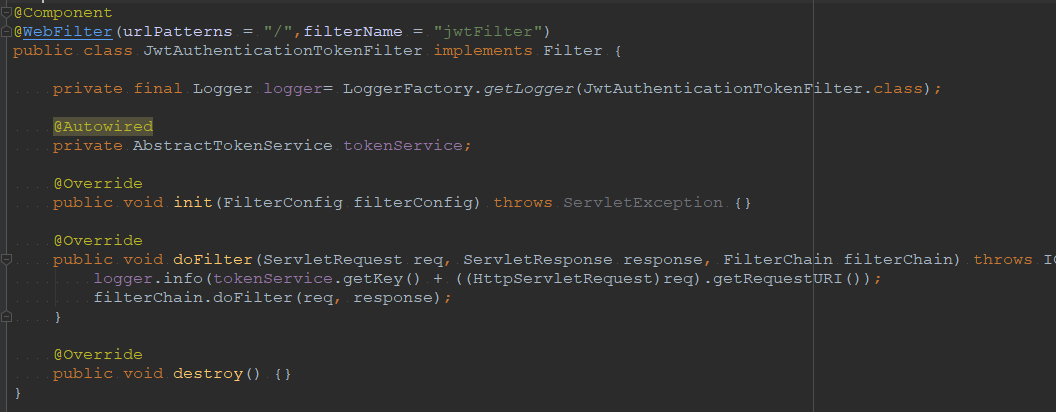
https://www.cnblogs.com/mooba/p/11276062.html

### 过滤器和拦截器

#### 过滤器

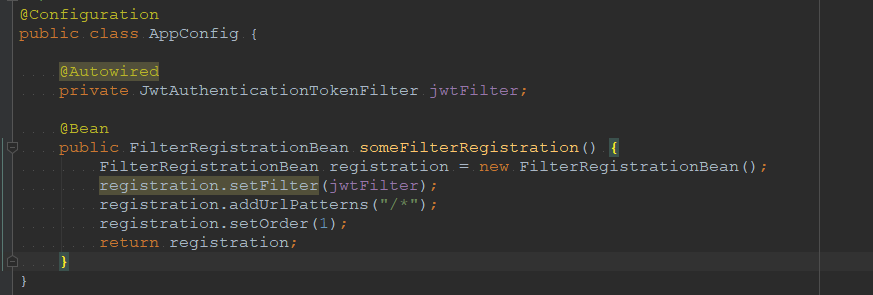
##### 方法1:

1. 实现Filter接口
2. 添加@WebFilter注解, 启动类使用@ServletComponentScan扫描@WebFilter、@WebServlet、@WebListener



##### 方法2:

1. 实现Filter接口
2. 注册该Filter (不使用@WebFilter注解和@Component注解)

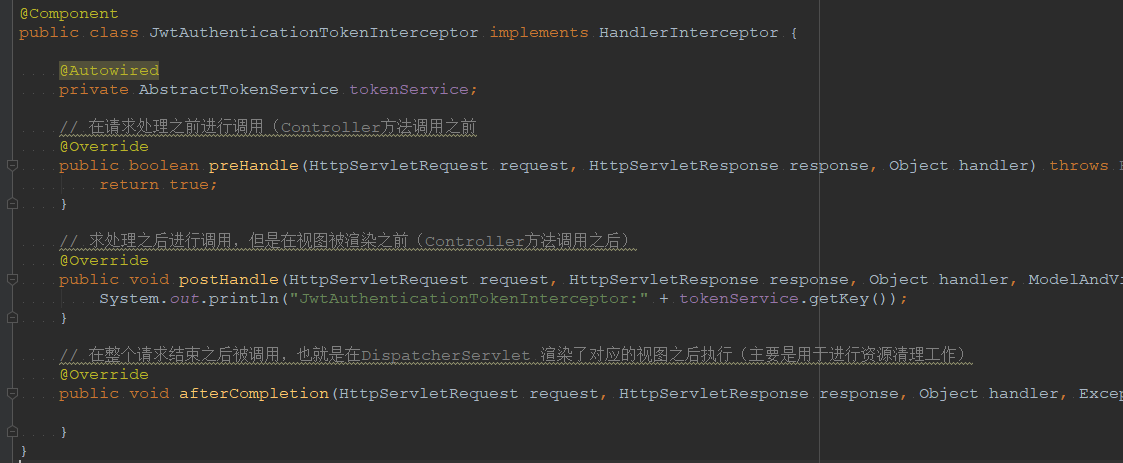


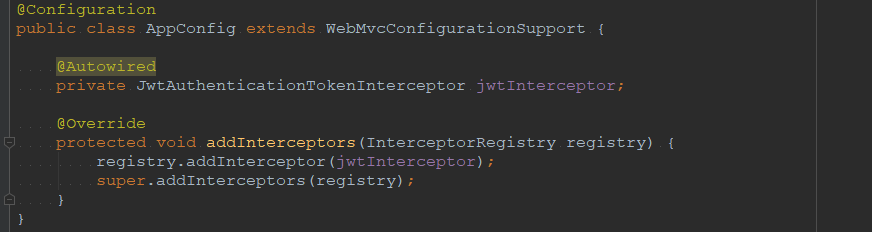
##### 方法3:

1. 实现Filter接口
2. 使用@Component注解即可, 默认过滤/\* . 可以和@WebFilter一起使用. 此时也可以使用Ioc容器中的bean
3. 通过@Order 进行排序

#### 拦截器

1. 实现接口
2. 进行注册 (必须要注册了)





#### 区别

### Bean 的生命周期

Spring bean 定义: 被Spring 容器管理的Java对象。

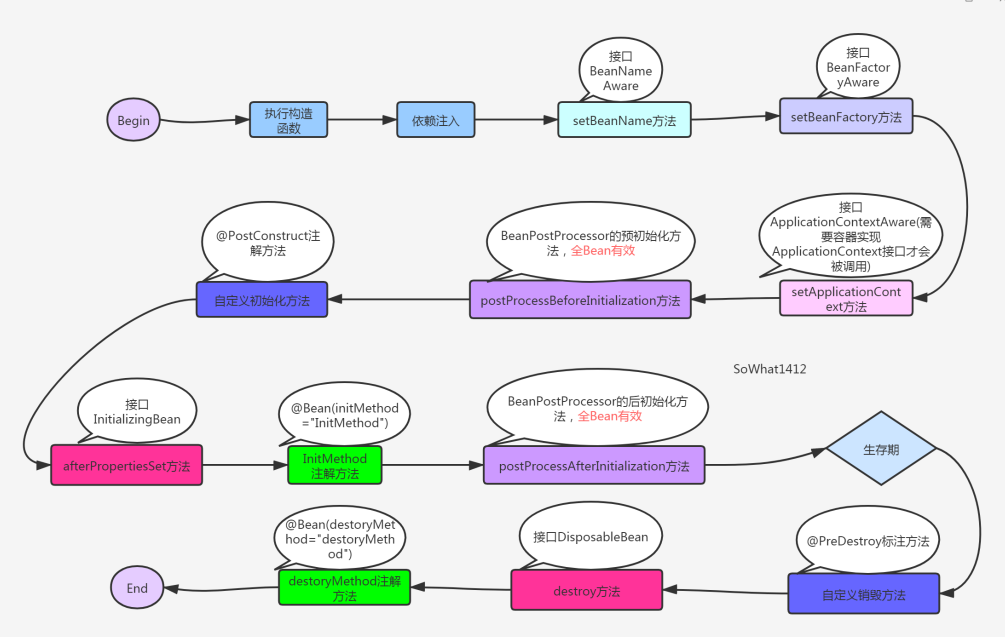
Spring 容器定义: 一个实现了ApplicationContext接口的类的实例, 一个bean工厂, 用来管理Bean对象的。

#### 作用范围

|  |  |
| --- | --- |
| 作用域(scope) | 描述 |
| singleton | 单例 默认 |
| prototype | 多例 |
| request | 每次请求创建一个bean, 在一次请求中该bean是一个 |
| session | 一次会话中, 一个bean对应一个实例 |
| global-session | 全局的session中, 一个bean对应一个实例 |

**\*** request, session, global-session 基于web的Spring ApplicationContext情形下有效.

#### 生命周期



\* init-method 中使用的方法若是@PostConstruct注解的方法或者InitializingBean接口的方法, 则该方法不会被调用。

\* 若该Bean是个后置处理器, 即实现了BeanPostProcessor接口, 那么该bean的生命周期中不会调用到它自己实现的后置处理器的方法.

\* 当使用@Configuration进行注入Bean时, 该Bean的执行顺序是 BeanFactoryAware ->BeanNameAware-> BeanFactoryAware ... 后面是一样的, 就是Aware的调用不一样。

##### 2.2.1 后置处理器

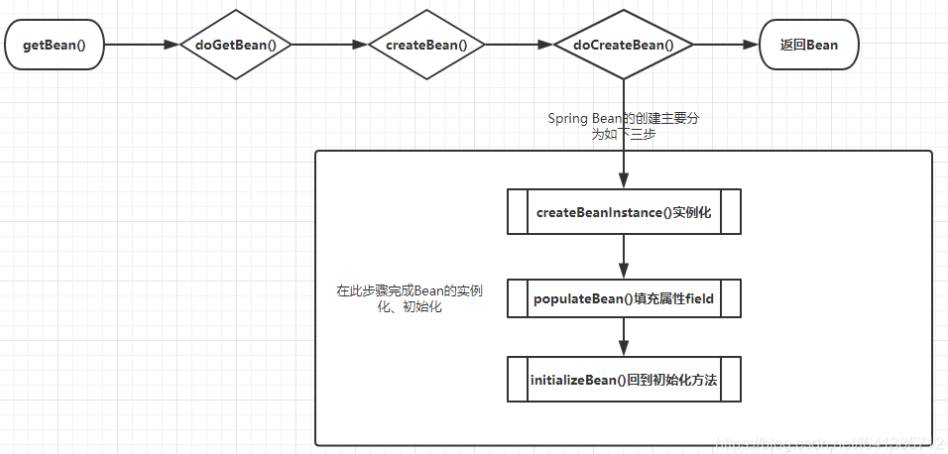
BeanPostProcessor 后置处理器, 有两个方法, postProcessBeforeInitialization在对象实例化之后初始化之前执行, postProcessAfterInitialization在对象初始化完成, 使用之前执行。

该处理起在BeanFactory容器中需要手动注册才能使用, 在ApplicationContext容易中会自动完成注册.

该处理器的postProcessBeforeInitialization方法返回null那么@PostConstruct注解的方法将不会被执行

多个处理器将按照order顺序进行执行

##### 2.2.2 Bean的创建流程



##### 2.2.3 循环依赖

对于单例的bean, spring 在创建的时候使用了3级缓存

1. singletonFactories: 单例对象工厂cache
2. earlySingletonObjects: 提前曝光的单例对象cache, 原始的bean, 尚未设置属性
3. singletonObjects: 单例对象的cache, 已经完全初始化好了, 可以直接使用

为什么不使用一级缓存, 因为一个缓存中存在已创建好的, 创建一半的, 多线程常见一半的可能被使用

为什么不用二级缓存, 而用三级? 不清楚 有的是说aop 有的是说扩展 有的说提升速度

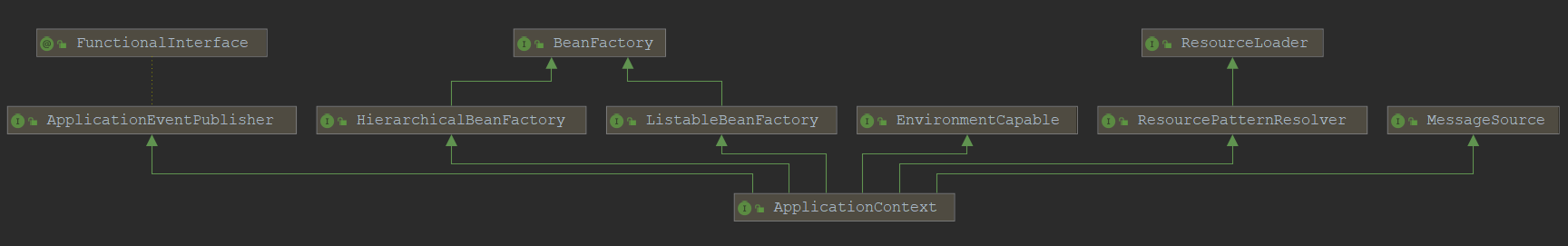
### Bean工厂

#### BeanFactory

1. 顶层工厂接口
2. 懒加载 获取bean的时候才会实例化bean
3. 后置处理器BeanPostProcessor 需要注册才能生效

#### ApplicationContext

1. BeanFactory的子类
2. 既有管理bean的功能又集成了其它spring的综合能力
3. 国际化 MessageResource
4. 事件机制 ApplicationEventPublisher
5. 资源访问 ResourcePatternResolver
6. 对web支持 EnvironmentCapable
7. Bean 的管理且可以自动装载, 例如: BeanPostProcessor
8. 预加载, 提前实例化



# Spring boot Security

Authentication: 鉴权

Authorization: 授权

UserDetails

Security中的用户信息.

UserDetailsService接口

UserDetailsServiceManager接口

InMemoryUserDetailsManager和JdbcUserDetailsManager实现类

用来操作UserDetails的。

WebSecurityConfigurerAdapter

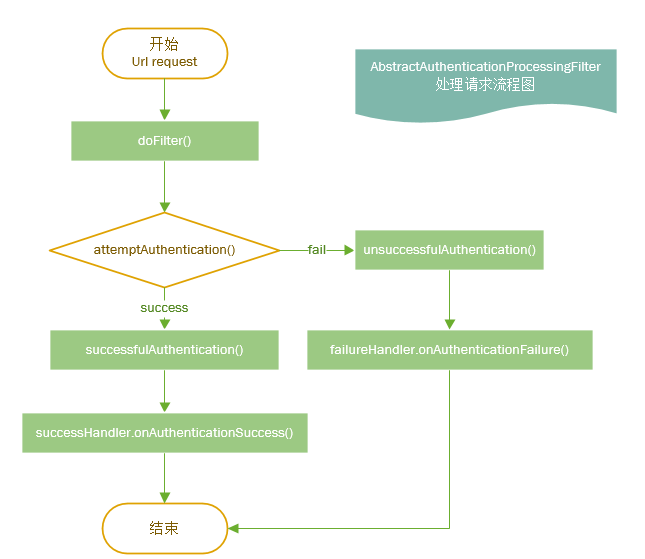
用来配置授权的

AbstractAuthenticationProcessingFilter

是处理 form 登陆的过滤器, 与 form 登陆有关的所有操作都是在该类及其子类中进行的。UsernamePasswordAuthenticationFilter就是它的一个子类.

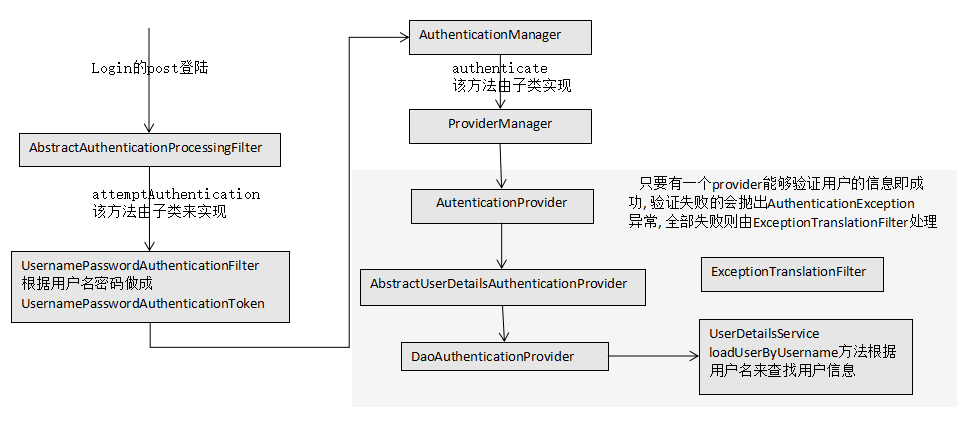
AuthenticationSuccessHandler为授权成功的处理类。

AuthenticationFailureHandler为授权失败的处理类。

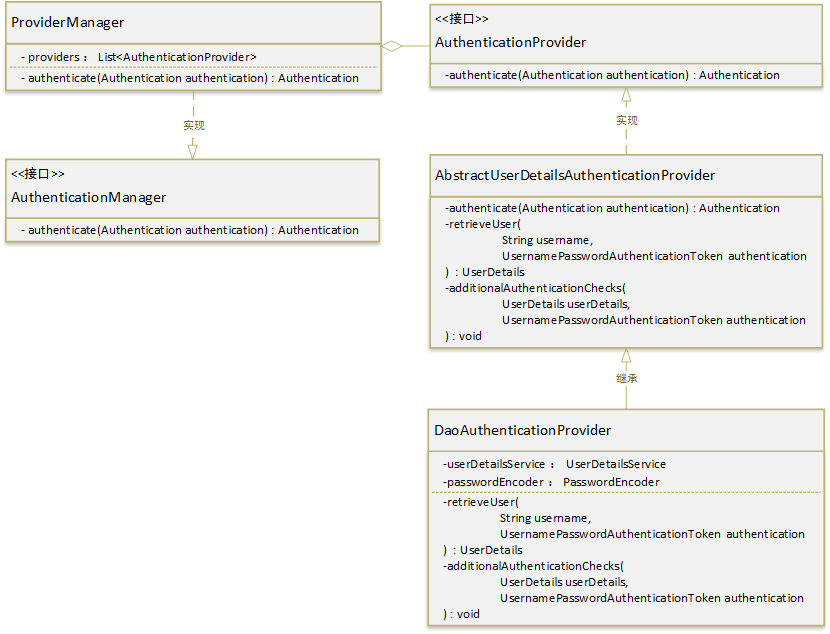
附件: <https://www.jianshu.com/nb/2455141>

https://blog.csdn.net/yuanlaijike/category\_9283872.html

认证的流程



类之间的关系



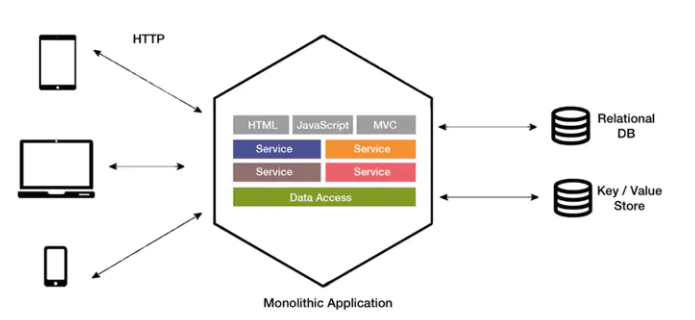
# Spring cloud

## SOA

SOA代表面向服务的架构, 将应用程序根据不同的职责划分为不同的模块, 不同的模块直接通过特定的协议和接口进行交互。使整个系统切分成很多单个组件服务来完成请求, 当流量过大时通过水平扩展相应的组件来支撑, 所有的组件通过交互来满足整体的业务需求。

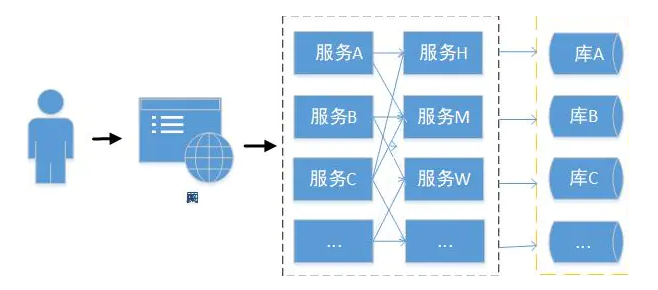
服务化架构是一套松耦合的架构, 服务的拆分原则是服务内部高内聚, 服务之间低耦合。

### 传统的单体架构



小型项目好维护管理, 业务复杂发版维护难度增加.

### 服务化架构



更易维护, 更高的可用性, 更好的伸缩性.

## 微服务

微服务架构是 SOA 架构思想的一种扩展, 更加强调服务个体的独立性、拆分粒度更小。

简单的来说微服务架构风格想要开发一种由多个小服务组成的应用。 每个服务运行于独立的进程, 并且采用轻量级交互。 多数情况下是一个http的资源api。 这些服务具备独立的业务能力并且可以通过自动化部署方式独立部署。 这种风格使最小化集中管理, 从而可以使用多种不同编程语言和数据存储技术。

服务独立、扩展性好、可靠性强, 但同时, 也面临一些新的问题, 比如运维复杂性, 分布式复杂性、监控复杂性等等。

相比于SOA, 微服务微服务需要彻底组件化和服务化, 可以独立的对外提供服务, 不在强调SOA架构里比较重要的ESB企业服务总线, 强调每个微服务有自己的运行空间, 包括数据库资源, 微服务的切分粒度更小。

## 并发系统的三大剑客

缓存中间件(redis), 消息中间件(rabbitMq), 通讯中间件(RPC)

## Spring cloud简介

Spring Cloud是基于Spring Boot的一整套实现微服务的框架。它提供了微服务开发所需的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等组件。最重要的是，基于Spring Boot, 会让开发微服务架构非常方便。

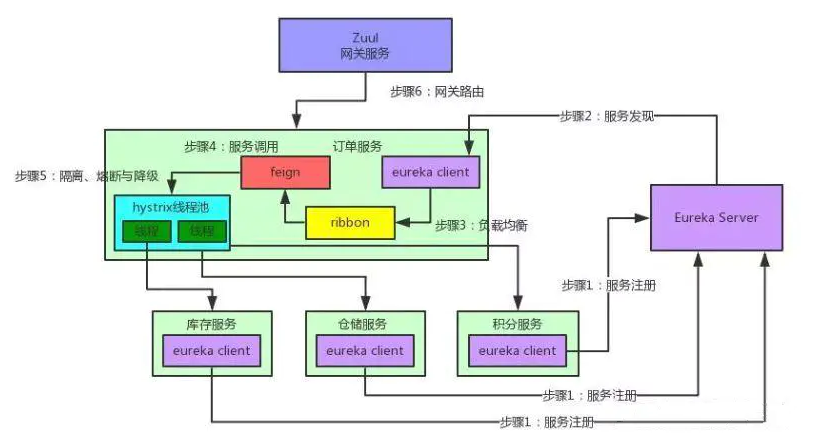
Spring Cloud是一套框架, 是一个解决微服务架构实施的综合性解决框架, 整合了诸多被广泛实践和证明过的框架作为基础部件, 大量的兼容性测试保证了更好的稳定性。 极高的社区活跃度。

Dubbo专注于服务治理, Spring cloud关注于微服务架构生态。



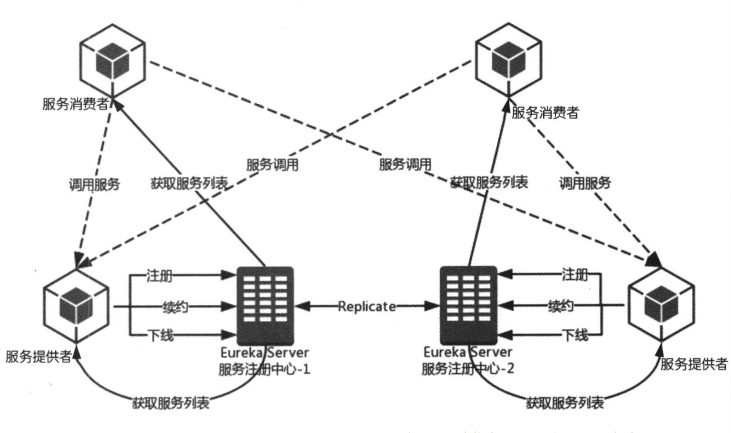
## 核心组件

Netflix 公司提供了包括Eureka, Hystrix, Zuul, Archaius在内的很多组件, spring在Netflix的基础上封装了一些列组件命名为spring cloud Eureka, spring cloud Hystrix等。



### Eureka

注册中心, Spring cloud Eureka 服务注册与发现器。 包括服务注册中心(类似于房产中介), 服务提供着(类似开发商), 服务消费者(购房者)。



1. 两台Eureka服务注册中心构成的服务注册中心的主从复制集群
2. 然后服务提供者向注册中心进行注册、续约、下线服务等
3. 服务消费者向Eureka注册中心拉去服务列表并维护在本地(这也是客户端发现模式的机制体现！)
4. 然后服务消费者根据从Eureka服务注册中心获取的服务列表选取一个服务提供者进行消费服务。

### Ribbon

负载均衡, 消费者根据Eureka服务注册中心获取的列表来选择一个服务者提供的服务进行消费, 如何选择即通过spring cloud ribbon 来实现。 即客户端负载均衡器。



当消费者找到多个满足需求的服务提供者时:

1. 优先选择一个zone(空闲)且负载较少的Eureka Server进行链接.
2. 定期从Eureka更新, 过滤服务和实例列表.
3. 根据负载均衡策略(轮询, 随机Random, 根据相应时间加权, 自定义等), 从注册表中选择一个真正的实例地址.
4. 通过rest client对服务发起调用.

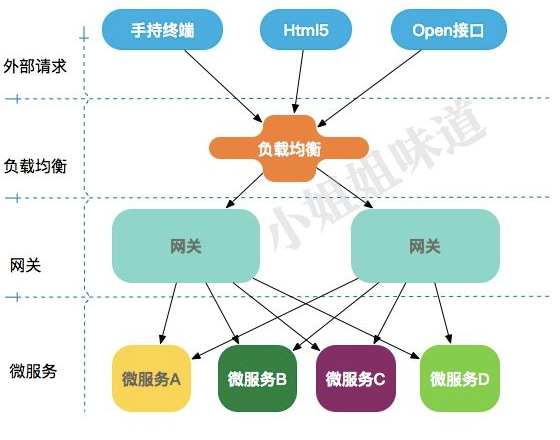


### Feign

服务调用, 用于简化Eureka和Ribbon的使用. 即RPC(远程调用)

### Hystrix, Resilience4j

熔断器, 类似于电路保护开关。 Hystrix对每个服务开了一个线程池, 并有比较复杂的规则, 来控制这些出问题的服务的行为。



### Zuul, Gateway

反向路由, 提供统一的入口, 屏蔽内部细节。

1. 安全认证, 提供统一的认证方式和鉴权功能, 避免重复开发.
2. 熔断, 限流。 针对服务问题, 进行熔断操作; 对流量进行预估, 限制访问.
3. 日志监控, 统一流量入口, 进行流量分析和监控.
4. 屏蔽内部细节, 对外提供一致的接口.
5. 实现灰度. 使用自定义策略实现分流, 达到测试的目的.

**各个组件学习**

附录: http://c.biancheng.net/view/5347.html

## Eureka

### 简介

组件EurekaServer注册中心. 微服务注册中心提供服务注册和服务发现, 方便微服务之间的相互调用. spring cloud是基于http协议的, 所以它的一个优势是可以不同语言之间进行交互, dubbo局限于java之间的通讯.

Eureka可以分为两个主体, Eureka server和Eureka client.

Eureka server：

1. 服务注册, 存储eureka client的注册信息, 用二层缓存机制来缓存整个注册表。
2. 提供注册表, eureka client从server拉去最新的注册表.
3. 同步状态, eureka client 通过心跳和注册与server同步客户端状态.

Eureka client:

1. 注册服务, 提供自身的ip, port, url等。
2. 服务续约, 每个30秒心跳续约, 90秒没用续约, server从注册表删除. 参数可配置.
3. Eviction 服务剔除, 没有心跳后, 从注册表删除.
4. Cancel 服务下线, client编程的方式调用下线。

DiscoveryManager.getInstance().shutdownComponent();

1. getRegisty 获取注册表信息, 会缓存到本地, 30秒更新一次, 默认使用压缩json来获取注册表信息。
2. Remote call 远程调用, 通过http请求调用注册的服务.

**Eureka具有高可用性, 但是其服务节点之间的数据可能是不一致的即具有非强一直性。**

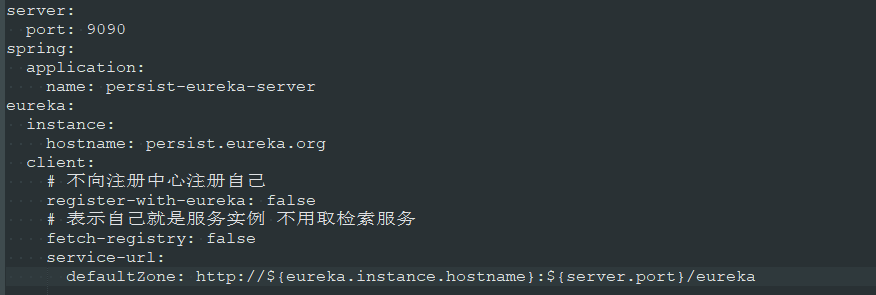
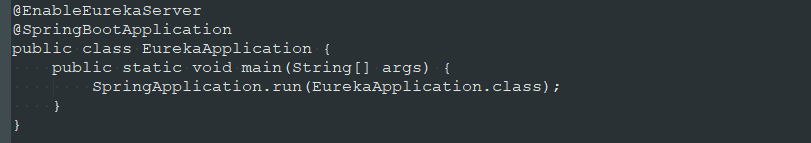
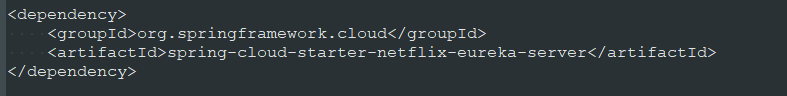
### 简单应用

#### 添加共通依赖

导入spring cloud 对应版本的依赖pom



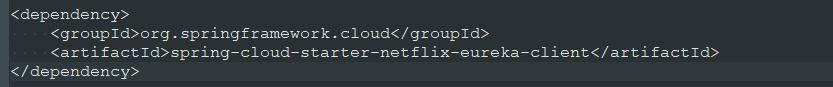
#### Server配置

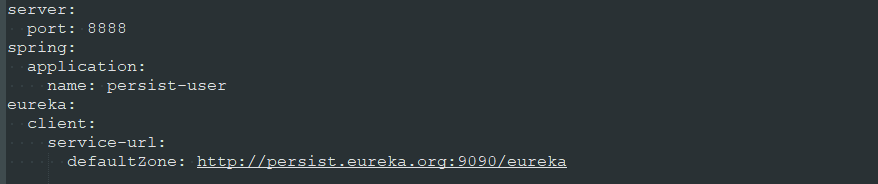


直接访问其localhost根路径 即可以看到注册表

#### Client配置

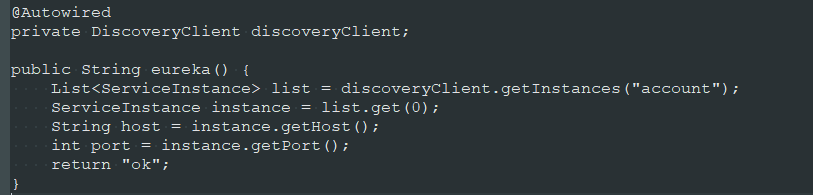
Provider和Consumer都当作client来配置





启动类添加 @EnableEurekaClient

1. 相互访问



即可以通过DiscoveryClient来获取服务, 然后取到host和port根据url调用了

### 配置参数

Spring.application.name # 系统名称

Server.port # 端口号

**Eureka.server**

Peer-eureka-nodes-update-interval-ms: # 节点更新间隔 默认60秒

Eviction-interval-timer-in-ms:60000 # 服务器清理无效节点间隔时间

Enable-self-preservation: false # 关闭自我保护模式, 默认true

Renewal-percent-threshold: 0.85 # 自我保护的错误系数, 默认是0.85

Wait-time-in-ms-when-sync-empty: 300000 # 同步失败的等待时间, 默认5分钟

Number-of-replication-retries: 5 # 同步失败的从试次数

**Eureka.client**

serviceUrl.defaultZone: http://xxx/xxx # 默认注册中心的地址, 微服务启动的时候去寻找, 注册.多个地址用“,” 号分隔。

Register-with-eureka: true # 允许自己注册自己

Fetch-registry: true # 是否从获取信息 从其它节点上

Registry-fetch-interval-seconds:30 # 客服端拉取服务注册信息间隔

Health-check.enabled: true # 是否启用客户端健康检查

eureka-server-connect-timeout-seconds: 5 # 连接Eureka服务器的超时时间，单位：秒，缺省：5

eureka-server-read-timeout-seconds = 8 # 从Eureka服务器读取信息的超时时间，单位：秒，缺省：8

filter-only-up-instances = true # 获取实例时是否只保留状态为 UP 的实例，缺省：true

eureka-connection-idle-timeout-seconds = 30 # Eureka服务端连接空闲时的关闭时间，单位：秒，缺省：30

eureka-server-total-connections = 200 # 从Eureka客户端到所有Eureka服务端的连接总数，缺省：200

eureka-server-total-connections-per-host = 50 # 从Eureka客户端到每个Eureka服务主机的连接总数，缺省：50

**Eureka.instance**

Appname: xxx # 服务名 默认取 spring.application.name, Applocation显示名称

Hostname: localhost # 应用主机名, 当Prefer-ip-address 为false时 status显示的连接为它

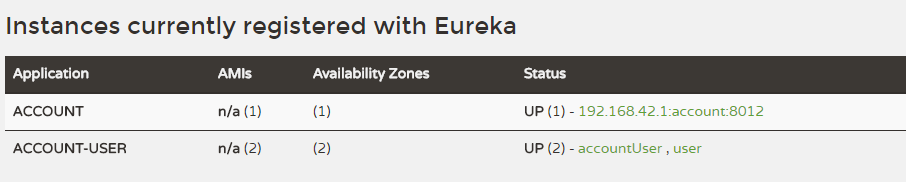
Prefer-ip-address: true # 客户端注册时使用自己的Ip而不是主机名 默认false

Lease-expiration-duration-in-seconds: 90 # 服务失效时间, 失效后将被剔除

Lease-renewal-interval-in-seconds: 30 # 服务续约时间(心跳)频率, 默认30秒

Ip-address: 127.0.0.1 # 设置ip地址, status 显示的服务连接ip地址

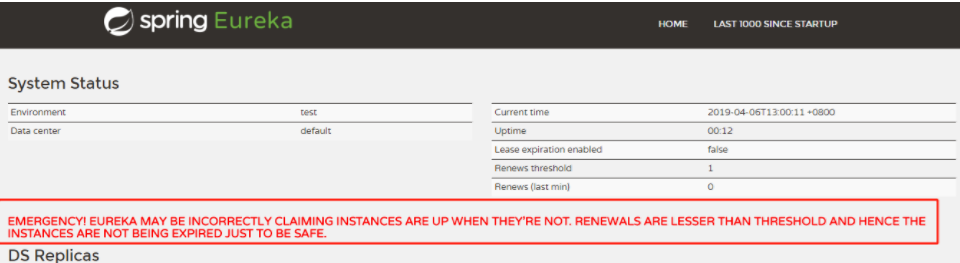
Instance-id: # 设置注册服务中心显示的实例名称, 默认为ip : aplicationName : port, 显示在status栏目处。



**附录： https://www.cnblogs.com/exmyth/p/13489627.html**

#### 自我保护机制

默认开启自我保护机制, 即当检查到client失联那么会从注册表剔除, 但是当超过默认1-0.85的client失联时, 会触发自我保护机制。 如下图：



自我保护机制:

1. Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
3. 当客户端心跳恢复时，Eureka 会自动退出自我保护机制。当前实例新的注册信息会被同步到其它节点中

对应的配置:

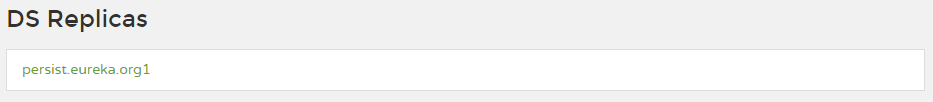
eureka.server.enable-self-preservation=true # 开启或者关闭

#### 缓存机制

附录 ： https://www.cnblogs.com/yixinjishu/p/10871243.html

#### 集群配置

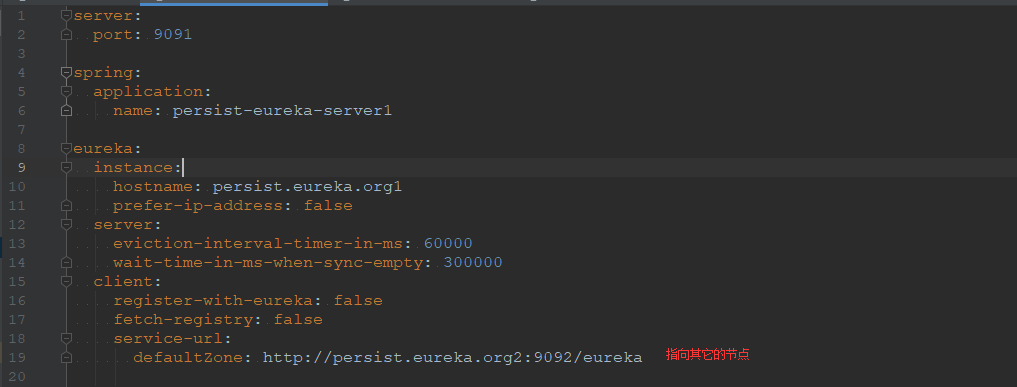
集群配置主要就是将 defaultZone 指向其它的服务节点即可, 配置成功后将看到如下:



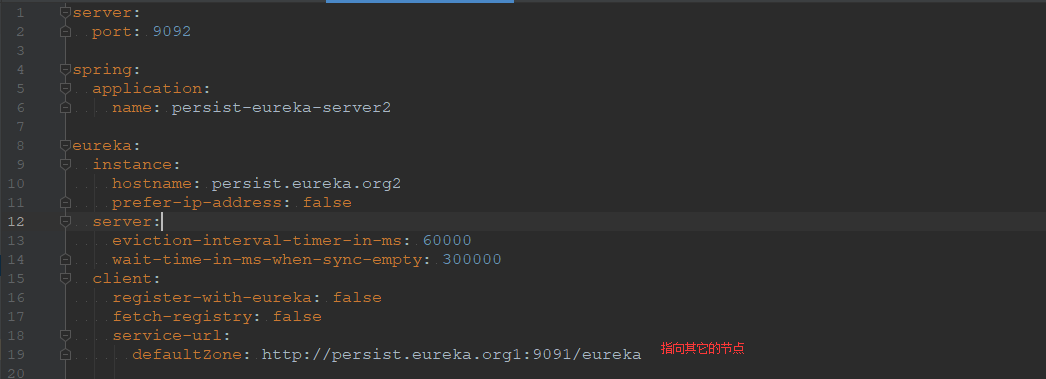
client端的 defaultZone 指向所有节点的地址即可。

**下面时节点的配置参照:**

**节点1:**



**节点2**



### Ribbon

#### 简介

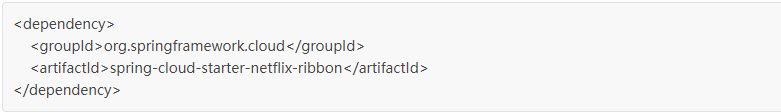
Spring Cloud Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡工具，它基于 Netflix Ribbon 实现。通过 Spring Cloud 的封装，可以让我们轻松地将面向服务的 REST 模版请求自动转换成客户端负载均衡的服务调用。

负载均衡器, 主流的负载均衡有两种

1. 集中式负载均衡，在消费者和服务提供方中间使用独立的代理方式进行负载，有硬件的(比如 F5)，也有软件的(比如 Nginx)。
2. 客户端自己做负载均衡，根据自己的请求情况做负载，Ribbon 就属于客户端自己做负载。

\* Ribbon 可以那里单独使用就如同spring一样, 下面是结合了spring boot 的spring cloud ribbon .

#### 依赖引入



其实不用配置, 因为eureka已经引入了ribbon 直接用就行.

#### 配置参数

# 开启 Ribbon 的饥饿加载模式。默认是懒汉式

ribbon.eager-load.enabled=true

#指定需要饥饿加载的服务名，也就是你需要调用的服务，若有多个则用逗号隔开。

ribbon.eager-load.clients=ribbon-eureka-demo

# 指定自定义的负载均衡策略

ribbon-config-demo.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RetryRule

# 禁用 Eureka 禁用了 Eureka 之后，就不能使用服务名称去调用接口了，必须指定服务地址。

ribbon.eureka.enabled=false

# 禁用 Eureka 后手动配置服务地址

ribbon-config-demo.ribbon.listOfServers=localhost:8081,localhost:8083

# 请求连接的超时时间

ribbon.ConnectTimeout=2000

# 请求处理的超时时间

ribbon.ReadTimeout=5000

# 也可以为每个Ribbon客户端设置不同的超时时间, 通过服务名称进行指定：

ribbon-config-demo.ribbon.ConnectTimeout=2000

ribbon-config-demo.ribbon.ReadTimeout=5000

# 最大连接数

ribbon.MaxTotalConnections=500

# 每个host最大连接数

ribbon.MaxConnectionsPerHost=500

# 为 Ribbon 指定对应的配置

<clientName>.ribbon.NFLoadBalancerClassName: Should implement ILoadBalancer(负载均衡器操作接口)

<clientName>.ribbon.NFLoadBalancerRuleClassName: Should implement IRule(负载均衡算法)

<clientName>.ribbon.NFLoadBalancerPingClassName: Should implement IPing(服务可用性检查)

<clientName>.ribbon.NIWSServerListClassName: Should implement ServerList(服务列表获取)

<clientName>.ribbon.NIWSServerListFilterClassName: Should implement ServerList­Filter(服务列表的过滤)

#### 负载均衡策略

默认是轮询, 策略接口 IRule

**1) BestAvailabl**

选择一个最小的并发请求的 Server，逐个考察 Server，如果 Server 被标记为错误，则跳过，然后再选择 ActiveRequestCount 中最小的 Server。

**2) AvailabilityFilteringRule**

过滤掉那些一直连接失败的且被标记为 circuit tripped 的后端 Server，并过滤掉那些高并发的后端 Server 或者使用一个 AvailabilityPredicate 来包含过滤 Server 的逻辑。其实就是检查 Status 里记录的各个 Server 的运行状态。

**3) ZoneAvoidanceRule**

使用 ZoneAvoidancePredicate 和 AvailabilityPredicate 来判断是否选择某个 Server，前一个判断判定一个 Zone 的运行性能是否可用，剔除不可用的 Zone（的所有 Server），AvailabilityPredicate 用于过滤掉连接数过多的 Server。

**4) RandomRule**

随机选择一个 Server。

**5) RoundRobinRule**

轮询选择，轮询 index，选择 index 对应位置的 Server。

**6) RetryRule**

对选定的负载均衡策略机上重试机制，也就是说当选定了某个策略进行请求负载时在一个配置时间段内若选择 Server 不成功，则一直尝试使用 subRule 的方式选择一个可用的 Server。

**7) ResponseTimeWeightedRule**

作用同 WeightedResponseTimeRule，ResponseTime-Weighted Rule 后来改名为 WeightedResponseTimeRule。

**8) WeightedResponseTimeRule**

根据响应时间分配一个 Weight（权重），响应时间越长，Weight 越小，被选中的可能性越低。

1. **自定义**

实现IRule接口, 主要是choose接口, 然后再配置文件中, 使用配置参数指定该策略, 也可以代码配置 .

ribbon-config-demo.ribbon.NFLoadBalancerRuleClassName=net.biancheng.ribbon\_eureka\_demo.rule.MyRule

#### 重试机制

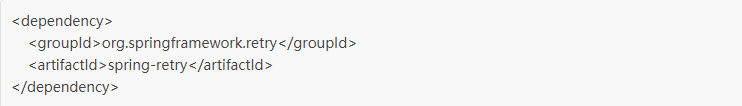
当请求失败时, 需要有重试机制保证服务的高可用。

1. **配置重试策略**

ribbon-config-demo.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RetryRule

1. **通过集成 Spring Retry 来进行重试操作**

**添加依赖**



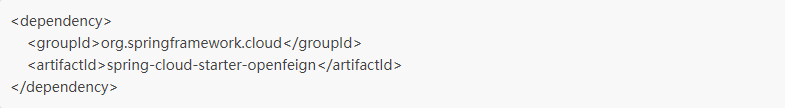
**配置**

# 对当前实例的重试次数  
ribbon.maxAutoRetries=1  
# 切换实例的重试次数  
ribbon.maxAutoRetriesNextServer=3  
# 对所有操作请求都进行重试  
ribbon.okToRetryOnAllOperations=true  
# 对Http响应码进行重试  
ribbon.retryableStatusCodes=500,404,502

### Feign

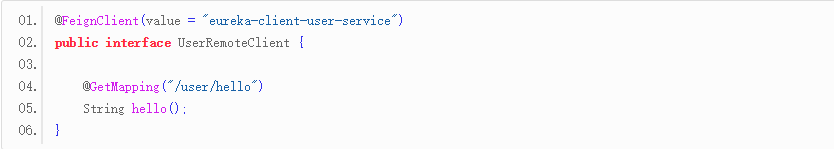
比HttpClient, Okhttp, HttpURLConnection, RestTemplate更简洁的一个声明式Rest客户端.

#### 依赖



#### 简单使用

1. 再启动类上添加 @EnableFeignClients.
2. @EnableFeignClients(basePackages="com.fangjia.api.client") 用于扫描接口所在包
3. @FeignClient(value = "eureka-client-user-service") 定义Feign 客户端



# MySql

## 范式

设计关系数据库时, 遵从不同的规范要求, 设计出合理的关系型数据库, 这些不同的规范要求被称为不同的范式, 各种范式呈递次规范, 越高的范式数据库冗余越小. 但是有些时候一昧的追求范式减少冗余, 反而会降低数据读写的效率, 这个时候就要反范式，利用空间来换时间。

目前关系数据库有六种范式, 第一范式(1NF)、第二范式(2NF)、第三范式(3NF)、巴斯-科德范式(BCNF)、第四范式(4NF)和第五范式(5NF，又称完美范式). 满足最低要求的范式是第一范式(1NF). 在第一范式的基础上进一步满足更多规范要求的称为第二范式(2NF), 其余范式以次类推. 一般说来, 数据库只需满足第三范式(3NF)就行了.

1. 第一范式

每列都要保证它的原子性, 也就是列不能再被分隔

1. 第二范式

唯一性, 有主键, 非主键字段依赖主键

1. 第三范式

非主键字段不能相互依赖, 不存在传递依赖

## 字符编码

Unicode(UCS): 字符编码的一种方法(规约) , 由国际组织设计。

utf8: 与ISO兼容, UTF(UCS Transformation Format)

ucs2: 就是用两个字节编码

\* 字符带ci的 例如utf8\_general\_ci 即为忽略大小写的意思。

### 默认编码

Mysql数据库的默认编码是latin1(ISO-8859-1的别名)

查看数据库的字符编码:

show variables like "char%"

### 设置编码

1 通过修改配置文件my.ini(windows下)或者/etc/my.cnf(linux下) 具体百度

2 创建数据库时设置编码

create database 数据库名 character set utf8;

3 创建表时设置编码

CREATE TABLE assign\_user (

yyuid int UNSIGNED NOT NULL COMMENT '用户的yyuid',

PRIMARY KEY (yyuid)

) ENGINE=InnoDB DEFAULT CHARSET=utf8;

4 设置字段编码

CREATE TABLE NewTable (

id varchar(255) CHARACTER SET utf8 NOT NULL,

PRIMARY KEY (id)

);

所以varchar(32) 这里的32是指的字符, 不是字节

## 外键

# 外键关联的字段必须是唯一的, 它可以是任意列或多列, 通常对应主表的主键, 下面用主键来表示外键关联的值。

1. 外键关联的字段类型必须保持一致.主键不允许null, 外键允许null。外键不为null时必须对应主键值。
2. 所有tables的引擎必须是InnoDb类型.
3. 建议创建表的同时创建外键约束, 删除时最好使用SET NULL约束类型.
4. 有主键的表为主表, 有外键的叫做从表. 可以一个表同时具有主外键叫自参照表。
5. Mysql 4.1.2后外键会自动建立索引。

### 创建

1.1 创建表时创建

constraint fk\_xx foreign key(xx) references xxx(xx) [on ...]

1.2 添加和删除

Alter table xxx add constraint fk\_xx foreign key(xx) references xxx(xx) [on ...];

Alter table xxx drop foreign key fk\_xx [on ...];

### 2. 外键约束类型

# 外键必须是主键的值, 若指定外键不存在于主键中, 那么不允许insert和update.

#### RESTRICT

主表删除时, 若从表有关联的记录, 则不允许父表删除。

#### No Action

同 restrict

#### CASCADE

主表delete, update的时候, 从表会delete, update掉关联的记录。

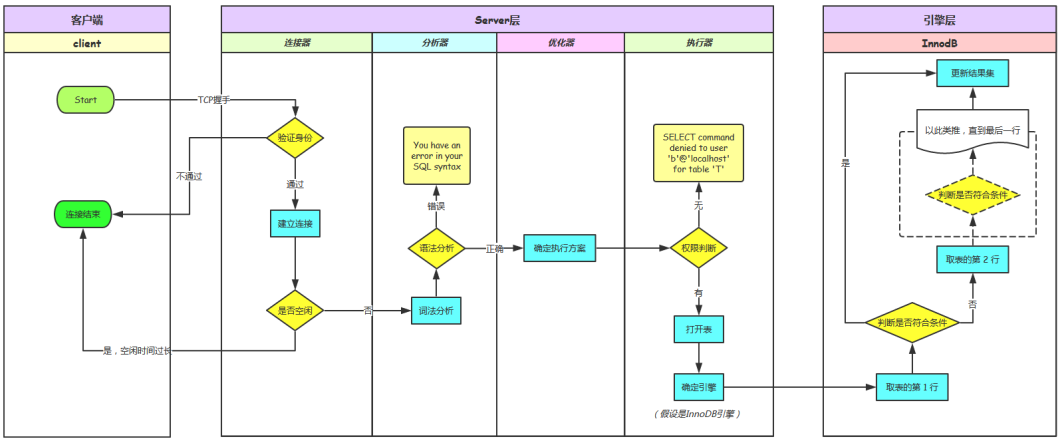
#### SET NULL

主表delete, update的时候, 从表会设置相应记录的外键为null。

On delete|update cascade|no action| set null|restrict

## SQL执行

### 执行流程



**连接器:** TCP握手后服务器来验证登陆用户身份，A用户创建连接后，管理员对A用户权限修改了也不会影响到已经创建的链接权限，必须重新登陆。

**查询缓存：**查询后的结果存储位置，MySQL8.0版本以后已经取消，因为查询缓存失效太频繁，得不偿失。

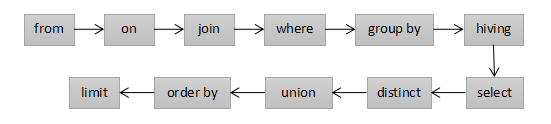
**分析器:** 根据语法规则，判断你输入的这个SQL语句是否满足MySQL语法。

**优化器:** 多种执行策略可实现目标，系统自动选择最优进行执行。

**执行器:** 判断是否有权限，将最终任务提交到存储引擎。

**存储引擎层:** 负责数据的存储和提取。其架构模式是插件式的，支持InnoDB、MyISAM、Memory等多个存储引擎。现在最常用的存储引擎是InnoDB，它从MySQL 5.5.5版本开始成为了默认存储引擎(经常用的也是这个)。

### 执行顺序



### 联合查询

对于多个表的联合查询并不是一定会按照固定的顺序去执行的, 而是会做一些优化, 以交叉处理的方式得到结果集.

具体: <https://blog.csdn.net/qq_27529917/article/details/78447882>

### SQL语句格式

SELECT

[ALL | DISTINCT | DISTINCTROW ]

[HIGH\_PRIORITY] [STRAIGHT\_JOIN] [SQL\_SMALL\_RESULT] [SQL\_BIG\_RESULT] [SQL\_BUFFER\_RESULT] [SQL\_CACHE | SQL\_NO\_CACHE] [SQL\_CALC\_FOUND\_ROWS]

select\_expr [, select\_expr ...]

[

FROM table\_references [PARTITION partition\_list] [WHERE where\_condition]

[GROUP BY {col\_name | expr | position}, ... [WITH ROLLUP]]

[HAVING where\_condition]

[WINDOW window\_name AS (window\_spec)[, window\_name AS (window\_spec)] ...]

[ORDER BY {col\_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]

[LIMIT {[offset,] row\_count | row\_count OFFSET offset}]

[INTO OUTFILE 'file\_name' [CHARACTER SET charset\_name] export\_options | INTO DUMPFILE 'file\_name' | INTO var\_name [, var\_name]]

[FOR {UPDATE | SHARE} [OF tbl\_name [, tbl\_name] ...] [NOWAIT | SKIP LOCKED] | LOCK IN SHARE MODE]

]

附录: https://www.cnblogs.com/dabric/p/12389474.html

## 常用SQL

### 1.连接数据库

### 2.存在插入不存在修改

mysql -hHost –u用户名 –p密码 –P端口号 db [-e "执行的sql语句"]

CREATE DATABASE `mydb` CHARACTER SET utf8 COLLATE utf8\_general\_ci;

### 3.批量导出|导入

insert into table (字段1, 字段2) values(值1, 值2) on DUPLICATE key update 字段2 = values(字段2);

Mysql[dump] -hHost -u用户名 -p密码 -P端口 数据库名 [-d(仅导出表结构)] [表名] [>|<] 文件路径

### 4.常用语句

DROP TABLE IF EXISTS B;

CREATE TABLE B LIKE A;

INSERT INTO B SELECT \* FROM A;

### 5.日期格式

DATE\_SUB(curdate(), INTERVAL -1 DAY)

DATE\_FORMAT(now(), '%Y-%m-%d')

SELECT DATE\_FORMAT(now(), '%H-%i-%s')

默认值 CURRENT\_TIMESTAMP 一个表中不要出现两次

### 插入查询的

INSERT INTO 表1(字段1, 字段2, 字段3) SELECT 字段1, 字段2, 字段3 FROM 表2 WHERE ...

### 当前的链接信息

Show [full] processlist; 默认只显示100个

### Count函数的使用

Count(\*) 包含了所有的列, 相当于行数在统计结果时不会忽略NULL, 它会自动优化到合适的列

Count(1) 会统计表中的所有的记录数, 不会忽略NULL, 包含字段为null的记录

count(列名)会统计该列字段在表中出现的次数, 会忽略字段为null的情况, 即不统计字段为null的记录

若列名为主键, count(列名)会比count(1)快

若列名不为主键, count(1)会比count(列名)快

若表多个列并且没有主键, 则 count(1) 的执行效率优于 count(\*)

若表有主键, 则 select count(主键)的执行效率是最优的

若表只有一个字段, 则 select count(\*)最优

实际业务中一般用count(1)比较普遍, 但是如果需要聚合多个列, 则用count(列名)比较合适

少量数据不忽略null选count(1),大量数据选count(\*)，忽略null选count(字段)

### Count统计计数

# 使用count()函数实现条件统计的基础是对于值为NULL的记录不计数

select count(num > 200 or null) from a;

select count(if(num > 200, 1, null)) from a;

select count(case when num > 200 then 1 end) from a;

### 拼接函数

# group\_concat(column\_name); 多条记录合成一条记录

# find\_int\_set(column\_name, strlist) 在多条记录中查询特定的列

SELECT GROUP\_CONCAT(i\_id) FROM city WHERE p\_id='2'

SELECT \* FROM city WHERE FIND\_IN\_SET(p\_id,'1,4')

### 函数和存储过程查询

SELECT USER(); #查看当前用户

#查询数据库中的函数或者存储过程

SHOW PROCEDURE|FUNCTION STATUS;

#查询数据库中的函数或存储过程

mysql.proc 表中存储相关信息

select \* from mysql.proc where db = 'girls' AND type = 'FUNCTION';

#查看存储过程或函数的创建代码

SHOW CREATE PROCEDURE|FUNCTION 库名.存储过程名|函数名;

### 树形结构查询

Oracle 可以使用 start with... connect by prior 这样的语句

MySql 使用：参照函数创建举例内容。

### Like 使用

Like '' 或者 not like ''

'\_' 用来匹配一个字符

'%' 用来匹配0个或者多个字符

like '%xiao/%%' escape '/'; /杠后的% 当作% 字符使用

[abc]:表示字符a或b或c

[!abc]:表示非a或b或c

### in和find\_in\_set

当参数是常量的时候用in , 变量的时候用find\_in\_set.

### any\_value

any\_value(字段), 允许非组字段的出现

## 存储过程和函数

# 创建

create procedure sp\_name ([proc\_parameter[,...]])

[characteristic ...]

begin

routine\_body

end;

create function sp\_name ([func\_parameter[,...]])

RETURNS type

[characteristic ...]

begin

routine\_body

end;

存储过程参数: [ IN | OUT | INOUT ] param\_name type

函数参数: param\_name type

type: mysql中的数据类型都可以

characteristic: 指定存储函数的特性

LANGUAGE SQL

| [NOT] DETERMINISTIC

| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }

| SQL SECURITY { DEFINER | INVOKER }

| COMMENT 'string'

routine\_body:

Valid SQL procedure statement or statements

# 删除

DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp\_name

# 使用

call 函数名或存储过程名();

在sql中直接使用类似于 sum() count()

在创建存储过程的时候一般都会用DELIMITER$$.....END$$ DELIMITER ;放在开头和结束，目的就是避免mysql把存储过程内部的";"解释成结束符号，最后通过“DELIMITER ;”来告知存储过程结束。

### 存储过程

1. 可以拥有输出参数
2. 需要call调用存储过程
3. 不允许包含return语句

### 函数

1. 不能有输出参数
2. 可以直接调用存储函数，不需要call语句
3. 必须包含一条return

举例:

delimiter $$

CREATE DEFINER=`root`@`localhost` FUNCTION `getSubRole`(id int) RETURNS varchar(1000) CHARSET utf8

begin

declare ptemp varchar(1000);

declare ttemp varchar(1000);

set ttemp = '#';

set ptemp = cast(id as char);

while ptemp is not null do

select group\_concat(sid) into ptemp from role where find\_in\_set(pid, ptemp);

if ptemp is not null then

set ttemp = concat(ttemp,",",ptemp);

end if;

end while;

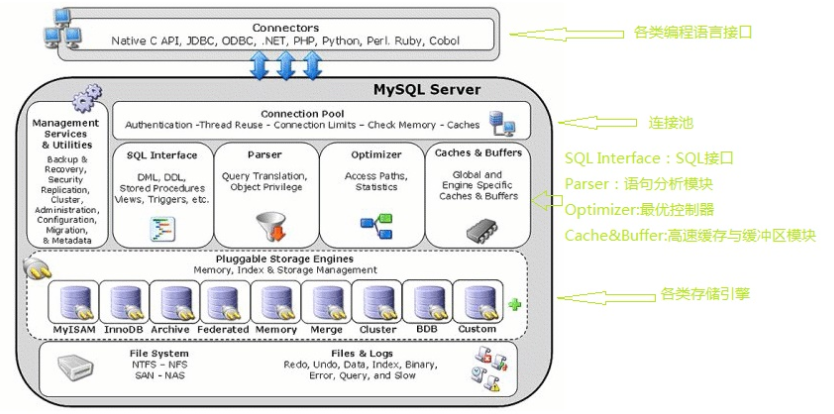
return ttemp;

end $$

delimiter;

## 存储引擎

Mysql5.1之后引入插件是存储引擎。



常用的存储引擎:

* MyISAM：默认的MySQL插件式存储引擎，它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。注意，通过更改STORAGE\_ENGINE配置变量，能够方便地更改MySQL服务器的默认存储引擎。
* InnoDB：用于事务处理应用程序，具有众多特性，包括ACID事务支持。
* BDB：可替代InnoDB的事务引擎，支持COMMIT、ROLLBACK和其他事务特性。
* Memory：将所有数据保存在RAM中，在需要快速查找引用和其他类似数据的环境下，可提供极快的访问。
* Merge：允许MySQL DBA或开发人员将一系列等同的MyISAM表以逻辑方式组合在一起，并作为1个对象引用它们。对于诸如数据仓储等VLDB环境十分适合。
* Archive：为大量很少引用的历史、归档、或安全审计信息的存储和检索提供了完美的解决方案。
* Federated：能够将多个分离的MySQL服务器链接起来，从多个物理服务器创建一个逻辑数据库。十分适合于分布式环境或数据集市环境。
* Cluster/NDB：MySQL的簇式数据库引擎，尤其适合于具有高性能查找要求的应用程序，这类查找需求还要求具有最高的正常工作时间和可用性。
* CSV : 引用由逗号隔开的用作数据库表的文件。
* Blackhole：用于临时禁止对数据库的应用程序输入。
* Example：可为快速创建定制的插件式存储引擎提供帮助。

### MyISAM

1. 存储数据产生三个文件, xxx.frm, xxx.MYD(数据文件), xxx.MYI(索引文件)

Myi文件保存数据的地址, 去myd文件查找

1. 5.5(<) 以前默认存储引擎 5.5(=<)以后默认InnoDb
2. 不同列的索引, 是平级的, 没有主次之分,

### InnoDb

1. 存储数据产生两个文件, xxx.frm, xxx.ibd

ibd数据文件, 采用聚集索引(主键), 进行数据的绑定。

数据查找时先找到主键的值, 然后在扫描找到物理上的存储位置, 找到数据, 即回表。

由于索引的实现数据结构是B+数, 所以每次索引的变化都会进行合并和分裂来保证绝对平衡, 从而对应的物理地址会发生变化, 所以回表的第二次仍然要扫描, 而不是直接存储地址。

1. InnoDb 中一切以主键位置, 非聚集索引只是优化加快了查询速度, 其本质还是依赖于主键。

## 索引

### 简介

存储于磁盘之上的一种数据结构。

1. 一个表最多有16个索引, 最大索引长度是256个字符(可以在编译mysql被改变)
2. 对于CHAR和VARCHAR的列可以使用前置索引, BLOB和TEXT必须使用列的前缀
3. 组合索引最多由15个列组成
4. 索引名一般以 idx\_ 作为开头
5. Msql通常找最少行索引或者唯一行最多的索引(当同时多个索引可用时)

### 查看一个表的索引

Show Index From 表名;

### 结构

1. tree, R-tree, hash, full-text
2. Tree: myisam, bdb, innodb, ndb, archive仅支持geometry数据类型。

B-tree: myisam, innodb

Hash 适用于随机访问的场合, 查询每条数据时间几乎相同, 适用于, =, in, <=>, 不适用于范围查询, 不能排序。

Full-text: 仅archive支持, 只能用于char, char, text用来代替% \* 效率低问题。

附录: https://www.cnblogs.com/lijc1990/archive/2013/06/20/3146378.html

### 索引表介绍

|  |  |
| --- | --- |
| **列名** | **描述** |
| Table | 表名 |
| Non\_unique | 0 索引不能重复 1 索引可以重复 |
| Key\_name | 索引的名称, 名称相同说明是同一个索引(联合索引) |
| Seq\_in\_index | 索引中的序列号, 一般用于知道在联合索引中的位置 |
| Column\_name | 索引的列名 |
| Collation | 排序规则 A 升序 Null无分类 |
| Cardinality | 表示索引中唯一值得数目的估计值(重复值越少该值越大), 值越大联合查询时使用该索引机会越大, 值小要考虑是否值得创建该索引。可以统计更新(百度)。 |
| Sub\_part | 前置索引,长数据类型会使用索引前缀(列被编入索引的数量), 全列通常是NULL |
| Packed | 关键字如何被压缩 没有压缩NULL |
| Null | 用于显示索引列中是否包含 NULL 若列含有 NULL, 该列的值为 YES, 若没有，则该列的值为 NO |
| Index\_type | 显示索引使用的类型和方法(BTREE、FULLTEXT、HASH、RTREE, SPATIAL) |
| Comment | 显示评注 |
| Index\_comment |  |

附录：

Index\_type(可以在创建索引时进行设置, 默认BTREE):

> BTREE：适合连续读取数据

> RTREE：适合根据一条数据找附近数据

> HASH：适合随机读取数据

> FULLTEXT

> SPATIAL

### 索引的类型

#### 5.1 聚集索引

物理上连续, 一个表只能有一个, 一般是主键索引。 如果没有主键, 则会自动创建一个隐藏列, 索引的逻辑存储顺序决定了该行物理上的存储顺序。 聚集索引, 索引的叶子节点就是数据节点。

#### 5.2 非聚集索引

除了聚集索引的其它索引。 逻辑上连续, 物理上不连续。即不影响物理存储顺序且可以有多个。索引的叶子节点仍然是叶子节点, 它有一个指针指向对应的数据块。

#### 5.3 其它

Index: 基本索引, 没有任何限制

Unique Index: 唯一索引, 索引的列必须唯一, 允许空值, 如果是组合索引, 则列的组合必须唯一

PRIMARY KEY: 主键索引(聚集索引), 特殊的唯一索引, 不允许有空值, 主键索引只允许有一个(创建表时创建, ALTER与其他不一样)

组合索引: 多个字段上创建索引, 查询中使用创建时索引的第一个字段, 索引才会被使用。 组合索引遵循最左前缀集合

FULLTEXT INDEX: 全文索引, 用来查询文本中的关键词, 不直接与索引中的关键词相比较。配合match against使用

覆盖索引: 减少回表的操作, 直接一次索引查找得到。也即是在索引树种就可以得到。

### 索引重复(冗余)

同一个索引类型, 如果有两个或者以上的索引包含了相同的索引信息, 那么存在索引冗余。(不同的索引类型, 索引值不同)

### 添加 删除一个表的索引

CREATE INDEX 索引名 ON 表名(列名...)

ALTER TABLE 表名 ADD|Drop (PRIMARY KEY | UNIQUE | INDEX | FULLTEXT) 索引名 (列名...)

### 索引优化

1. Where子句中出现的列, join子句中出现的列
2. LIKE 'mich%' 会用到索引, '%like' 不会用到索引
3. Where的多个and中, 必须都是一个多列索引的key\_part且必须包含key\_part1。如果都是各自的单一索引的话, 只使用遍历最少行的索引。
4. 在where中多个or的条件, 每一个条件都必须是一个有效索引。
5. Order by后面的条件必须都是同一索引的属性, 并且排序必须一致。
6. 所有GROUP BY列引用同一索引的属性, 并且索引必须是按顺序保存其关键字的。
7. 对智能的扫描全表使用FORCE INDEX告知MySQL, 使用索引效率更高。
8. 定期ANALYZE TABLE tbl\_name为扫描的表更新关键字分布
9. 定期使用慢日志检查语句，执行explain，分析可能改进的索引
10. 条件允许的话，设置较大的key\_buffer\_size和query\_cache\_size的值(全局参数)，和sort\_buffer\_size的值(session变量，建议不要超过4M)
11. 隐式转化索引失效
12. 离散度高适用做索引
13. 少于\*代替列信息
14. 列适用的函数和中间表会导致索引失效

### (InnoDB)锁

#### 9.1 悲观锁

很悲观, 每次拿取数据时都认为有别的线程会对数据进行修改, 所以会给数据上锁. 核心就是不支持多并发, 是单线程操作. 通过抢占时间片的方式来抢锁的使用权, 把并发变成了串行.

适用于多写的场景. 保证线程安全和数据安全.

mysql的行锁, 表锁, 读锁, 写锁. java中的synchronized.

悲观锁是mysql自己实现了, 可以通过相关语法使用或者进行显示声明.

#### 9.2 乐观锁

很乐观, 拿数据的时候不会人为有其它线程对数据进行修改. 核心是支持多并发, 每个线程在不同的时间节点对数据做更新操作, 每次更新时候都会判断其他线程是否对数据做了更新.

1. version版本号机制

数据表中添加数据版本号, 更新数据后会使其版本号+1, A线程进行更新数据时, 读取数据时读取到版本号, 更新时会比较版本信息是否一致, 否则从新执行整个更新操作.

1. CAS算法机制

判断当前的内存值和之前取到的值是否相等, 若相等则用新值更新, 若不相等则重试, 即不断重试(自旋操作).

适用于多读的操作, 加大数据的吞吐量. 需要自己去实现.

#### 9.3 悲观锁·共享锁

一把锁有很多个钥匙.

#### 9.4 悲观锁·排它锁

同一个资源只能有一把锁.

#### 9.5 行锁

给一条记录加锁.

#### 9.6 表锁

给一个表加锁.

### 自增索引

通过对主键ID指定AUTO\_INCREMENT属性(非主键字段也可以指定), 从而使得该主键成为一个自增索引, 当插入Null时该字段自动生成下一个序列编号(从该字段最大的那个数开始自增).

**Update自增列的值**

|  |  |  |
| --- | --- | --- |
| **操作** | **MyISAM** | **InnoDb** |
| 修改成已存在的数值 | duplicate 报错 | duplicate 报错 |
| 修改为不存在的 < 当前最大 | OK, 自增开始值不变 | OK, 自增开始值不变 |
| 修改为不存在的 > 当前最大 | 从修改后的值开始自增 | 自增开始值不变, 自增到修改后的这个值 duplicate 报错 |

**自增介绍**

5.1 之后引入 innodb\_autoinc\_lock\_mode, 5.1之前默认是 traditional local mode 的自增方式. 插入语句分类为: simple insert, bulk insert, mixed insert.



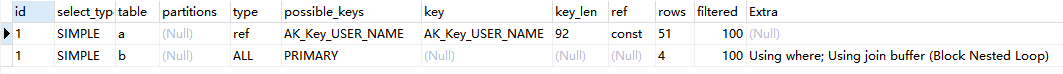
1. Traditional, 自增是连续的, 基于语句复制模式下的主从复制, 自增ID是一样的
2. Consecutive, 对于sample insert语句在开始时, 就会获得一个表锁, 获取到需要增加的ID的量后, 释放锁, 不用等到语句执行完成再释放锁. 对于其它的insert还是获取表锁, 语句执行完才释放, 主从和traditional一样
3. Interleaved, simple insert保证连续, bulk insert的Id可能会有空洞, 主从复制基于语句复制Id会不同, 只能基于行复制

### \* 其它

1. 字段需要设置默认值
2. 不同表相同属性 应保持一致性
3. SELECT COUNT(\*) 在没有where的语句中执行效率小于SELECT count(col\_name), 但有where它的效率快。
4. **列的离散度**: 离散型越高, 越适合作为索引 distinct(列名)/count(列名), 离散度低作为索引会成为冗余索引, 可能不会被使用。
5. Like 'xxx%', % 用到后面 放到前面将不会使用索引 '%xxx'.
6. **索引项比较规则**: 不同类型索引项的比较是不一样的, 高版本以上默认的**隐式转换**不会走索引.

## 执行计划Explain

查询语句前使用explain即可



执行计划表介绍:

|  |  |
| --- | --- |
| **列名** | **描述** |
| id | 选择标识符, sql的执行顺序(相同时由上到下执行, 不同时由大到小执行) |
| select\_type | 查询类型(SIMPLE, PRIMARY, UNION, DEPENDENT UNION, UNION RESULT, SUBQUERY, DEPENDENT SUBQUERY, DERIVED, UNCACHEABLE SUBQUERY) |
| table | 输出结果集的表(表名或者简称) |
| partitions | 匹配的分区 |
| type | 表的访问方式(ALL, index, range, ref, eq\_ref, const, system, null 性能差->好) |
| possible\_keys | 查询时可能使用到的索引 没有是null |
| key | 实际所使用的索引 |
| key\_len | 索引字段的长度(索引的最大可能长度 越短越好) |
| ref | 列与索引的比较 |
| rows | 扫描(估算)出的行数 |
| filtered | 按条件过滤的行的百分比 |
| Extra | 执行情况的描述和说明 (Using where, Using temporary, Using filesort, Using join buffer, Impossible where, Select tables optimized away, No tables used) |

\* 通过收集统计不可能存在的结果, 执行计划是没用的。

**附录:**

Select\_type:

SIMPLE: 简单的select, 不使用 union或子查询等

PRIMARY: 查询中包含子查询, 最外层的select被标记成PRIMARY

UNION: UNION中的第二个或后面的select语句

type:

all: 全表扫描

index: 从索引中全部扫描(索引文件比较小, all是从硬盘读取, 它们都是度全表)

range: 使用一个索引来检索给定范围的行, between < > in

ref: 非唯一索引扫描

eq\_ref: 唯一性索引扫描, 对于每个索引表中只有一行与之匹配

const: 通过索引一次找到, 相当于查一行

system: 表只有一行记录(等于系统表), const类型特例

Extra:

Using Index: 不用读取表中所有信息，仅通过索引就可以获取所需数据, 索引用于读取

Using Where: 不用读取表中所有信息，仅通过索引就可以获取所需数据, 索引用于查询

Using temporary: 使用临时表来存储结果集, 常见 group by; order by 没使用到索引(要优化)

Using filesort: 包含 order by 操作, 无法利用索引完成的排序操作称为"文件排序"(要优化)

Using join buffer: 表明使用了连接缓存, 比如说在查询的时候, 多表join的次数非常多，那么将配置文件中的缓冲区的join buffer调大一些(要优化)

Impossible where: where子句的值总是false (where语句不正确一般)

Select tables optimized away: 没有group By通过索引进行聚合函数的优化

No tables used: Query语句中使用from dual 或不含任何from子句

Distinct: 优化distinct操作, 找到第一个匹配的后停止寻找同样值得动作

### 附件

**执行计划Explain详解: <https://blog.csdn.net/why15732625998/article/details/80388236>**

## BinLog

### 概念

二进制日志文件（后缀 .00000\*），记录用户对数据库的操作事件，表结构变更DDL（CREATE ALTER），表数据修改DML(INSERT UPDATE DELETE)。

没有造成数据修改（SELECT SHOW RollBack）不会被记录（即，记录的是事务提交后的数据操作事件）。

是server层实现，所有引擎都可以使用binlog日志。

通过追加的方式写入，一个文件写满之后，新建一个继续写。

对于非事务的语句，每次执行成功后直接写入。对于事务，则是每次事务提交成功后一次性写入（一次提交对应一次记录，顺序记录）。

主要用于数据恢复（MySql binLog 工具），主从复制（MySql Replication），增量数据备份或迁移。

[binLog理解](https://www.cnblogs.com/rjzheng/p/9721765.html)

[binLog结构解析](https://www.jianshu.com/p/ea666baf0d82)

### 配置

在my.cnf（window是my.ini）中进行配置。

Linux可以使用  mysql --help|grep my.cnf 来查看配置文件位置，window通过执行 show variables like '%data%' 的datadir来找到mysql安装位置从而找到配置文件。

**相关配置参数**

log-bin=指定目录（/var/lib/mysql/mysql-bin）

server-id=集群Id要不一样（123454）

max\_binlog\_size 设置binlog文件大小，达到值后生成新的文件保存日志

binlog-format 指定日志格式，5.7.7之前默认statement，之后默认row

[开启binlog设置](https://blog.csdn.net/king\_kgh/article/details/74800513)

**Mysql相关查询语句**

show variables like '%log\_bin%'; 查询logbin是否开启

show binary logs; 查询二进制日志

show master status; 查看日志状态

flush logs; 刷新日志

show binlog events in 'binlogfile'; 查看binlog具体事件类型

[binLog日志配置](https://www.cnblogs.com/hongdada/p/10983768.html)

### 三种模式

#### 3.1 Row

日志中记录每一行数据被修改前和后的情况。Slave端根据数据修改记录进行相同的修改。

优点: 记录详细明确，精准还原数据，复制或数据恢复过程可以并发进行。

缺点:产生大量的日志， 并且也不知道具体做了什么操作。对于修改记录多，字段长度大的操作来说，记录时性能消耗会很严重。阅读时也需要特殊指令来读取数据。

场景: StreamSet（ 轻量级的引擎(工具)，用于实时传输数据） 要求该模式，阿里的canal组件。

#### 3.2 Statement

记录可能引起数据变化的sql语句。

优点: 减少日志量，IO消耗很底

缺点: 需要上下文信息，某些特定函数功能导致复制出错，比如uuid()函数，自增主键

#### 3.3 Mixed

row和statement的结合。

### 应用场景

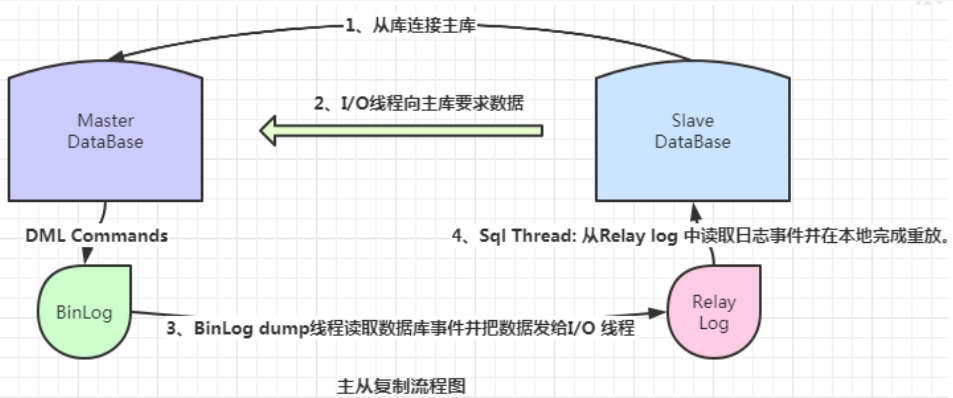
#### 4.1主从复制

主从复制通常用于，实时灾备，故障切换，读写分离，数据迁移。

##### 4.1.1 复制流程

1、主节点必须启用binlog日志，记录任何修改了数据库或数据的事件。

2、从节点开启一个线程（I/O Thread)把自己扮演成 mysql 的客户端，通过 mysql 协议，请求主节点的二进制日志文件中的事件 。



3、主节点启动一个线程（dump Thread），检查自己二进制日志中的事件，跟对方请求的位置对比，如果不带请求位置参数，则主节点就会从第一个日志文件中的第一个事件一个一个发送给从节点。

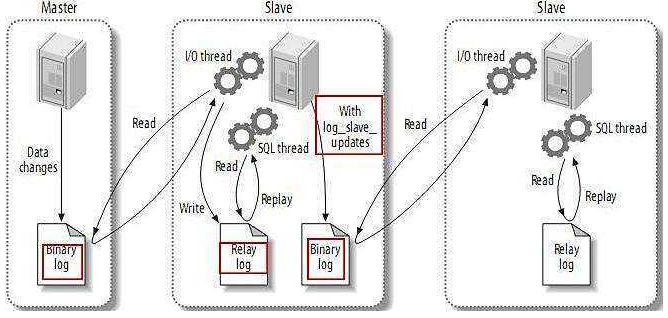
4、从节点接收到主节点发送过来的数据把它放置到中继日志（Relay log）文件中。并记录该次请求到主节点的具体哪一个二进制日志文件内部的哪一个位置（主节点中的二进制文件会有多个）。

5、从节点启动另外一个线程（sql Thread ），把 Relay log 中的事件读取出来，并在本地再执行一次。

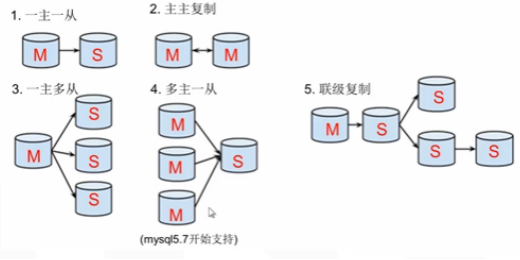
**总结**

从库，生成两个线程，一个I/O线程，一个SQL线程。I/O线程去请求主库的binlog，并将得到的binlog写到relaylog文件中，sql线程进行读取并解析成具体的操作，来实现主从数据的一致达到最终一致。

主库，生成一个log dump线程，用来给从库I/O线程传递binlog的。



##### 4.1.2 主从复制类型



##### 4.1.3 主从复制问题

MySql 3.23引入了Replication，没有relay log， 为实时同步复制，当从节点延时，会导致大量数据没备份到slave。

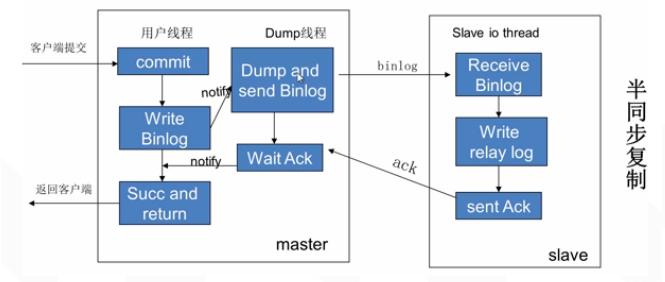
MySql 4.0.2添加了relay log异步复制，主库不用管从库的速度，但是当从库落后，主库宕机，从库升级为主库会导致部分数据丢失。

MySql 5.5引入半同步复制，即保证所有的从库中，只要有一个从库应答已经写到relay log中，就继续。

MySql 5.7.17全同步技术问世，所有的从节点都应答，才会继续。

\* 演进: 同步 -> 异步 -> 半同步 -> 全同步

MySql semi-sync（半同步复制）



[MySql 主从复制](https://cloud.tencent.com/developer/article/1544449)

#### 4.2 数据恢复

提前开启binlog日志的前提下，当发生数据丢失时可以通过mysql提供的命令进行数据恢复。具体使用用到时研究^\_^。

[binLog恢复MySql数据库](https://www.cnblogs.com/hanyifeng/p/5756462.html)

#### 4.3 增量数据迁移

数据迁移基本上都是 存量数据迁移，增量数据迁移，数据校验，切流。而增量数据迁移就可以使用监听binlog的方式实现。

例: 使用canal监听binlog迁移(https://www.iteye.com/blog/agapple-1796633)

## RedoLog

### 概念

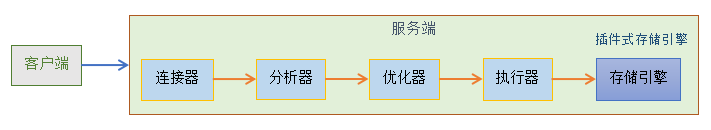
和BinLog是一样的，RedoLog也是用来记录数据库DDL，DML的操作事件的。

|  |  |  |
| --- | --- | --- |
| 日志文件 | BinLog | RedoLog |
| 日志类型 | 逻辑日志（存储了逻辑SQL修改语句） | 物理日志（存储了数据修改的值） |
| 实现层面 | MySql service实现的，所有引擎都可用 | InnoDb引擎实现的 |
| 文件大小 | 增量写入，写满了会在新建文件继续写 | 大小固定，可配置，循环写日志 |
| 适用场景 | 主从复制和数据恢复 | 崩溃恢复 |

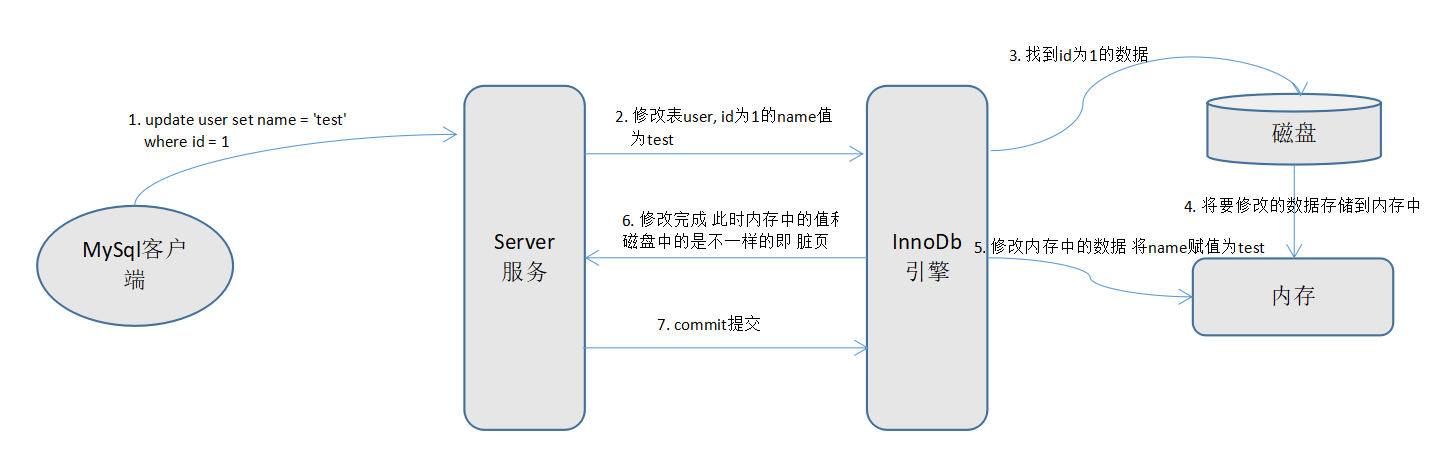
那么为什么有了BinLog，还需要搞个RedoLog呢？这就要看看InnoDb中数据是如何存储的了。

我们都知道，当执行一条sql语句时通常需要由客户端（java通常是jdbc），建立连接，连接到MySql服务端。有以下步骤。

1. 客户端，jdbc，navicat等
2. 连接器，校验用户信息
3. 存储缓存，字面意思，mysql8之后取消了
4. 分析器，词法分析，语法分析
5. 优化器，确定使用哪个索引，表连接顺序等
6. 执行器，是否具有执行权限，对引擎进行操作返回
7. 存储引擎，默认InnoDb，还有MyISAM、Memory等



RedoLog 就是用于执行器和存储引擎之间交互的。



当没有redolog时，执行器和存储引擎之间的交互大致如上，所有它存在的问题是:

1. MySql是以页为单位交换的（一页16kb），而一次事务可能只会修改很少一部分数据，那么这不是浪费资源么
2. 一个事务换可能修改多个数据页，这些数据页物理上不连续，那么每次都要随机IO一下，性能太差
3. 在进行6，7步骤的过程中，数据库宕机，这样就会造成部分数据丢失，内存中的就没了（没有落盘的内存中数据和磁盘上的数据是不一样的 此时称之为 **脏页**）

所以引入了redolog

1. 可以一次性将多个操作落盘到磁盘中，提高性能
2. 通过WAL（write-ahead logging）技术保证事务，WAL就是先写日志（物理日志，数据在哪里变成什么）后写磁盘（落盘）。
3. 由于先写日志的方式可以完成崩盘恢复

### 特点总结

上面已经知道了为什么会用到Redolog，这里总结下特点。

1. RedoLog和UndoLog属于InnoDb引擎的事务日志（就是和事务有关系），redolog用来恢复丢失数据（恢复到最后一次提交位置）也称之为前滚操作，undolog是用来回滚到之前的版本，称之为回滚操作。
2. RedoLog记录的是事务操作的变化，数据修改的值，不管事务是否提交都会记录。
3. RedoLog属于物理日志，它记录的是数据文件的物理变更，在什么位置，数据做了如何变更（数据库中每个页的修改）。
4. RedoLog使用WAL（日志优先落盘）来保证事务，也是使用这个技术来保证事务的4大特性的持久性。
5. 主要的应用场景是进行MySql的Crash Recovery（崩盘恢复）

### 配置介绍

InnoDb引擎至少有一个日志文件从做组，每个文件组至少有2个重做日志文件，如默认的ib\_logfile0和ib\_logfile1。

innodb\_log\_file\_size: 指定每个redo日志大小，默认值48MB

innodb\_log\_files\_in\_group: 指定日志文件组中redo日志文件数量，默认为2

innodb\_log\_group\_home\_dir: 指定日志文件组所在路劲，默认值./，指mysql的数据目录datadir

innodb\_mirrored\_log\_groups: 指定日志镜像文件组的数量，默认为1，此功能属于未实现的功能，在5.6版本中废弃，在5.7版本中删除了。

show variables like 'innodb%log%'; #列举RedoLog日志组的配置

### 工作流程简介

RedoLog的记录方式如图，它是循环写的。write pos可以看作是头，CheckPoint可以看作是尾。w->c是空闲的区域，c-w是已经写满的区域。写入时挪动w，落盘时挪动c，当w=c的位置时需要等待去落盘。

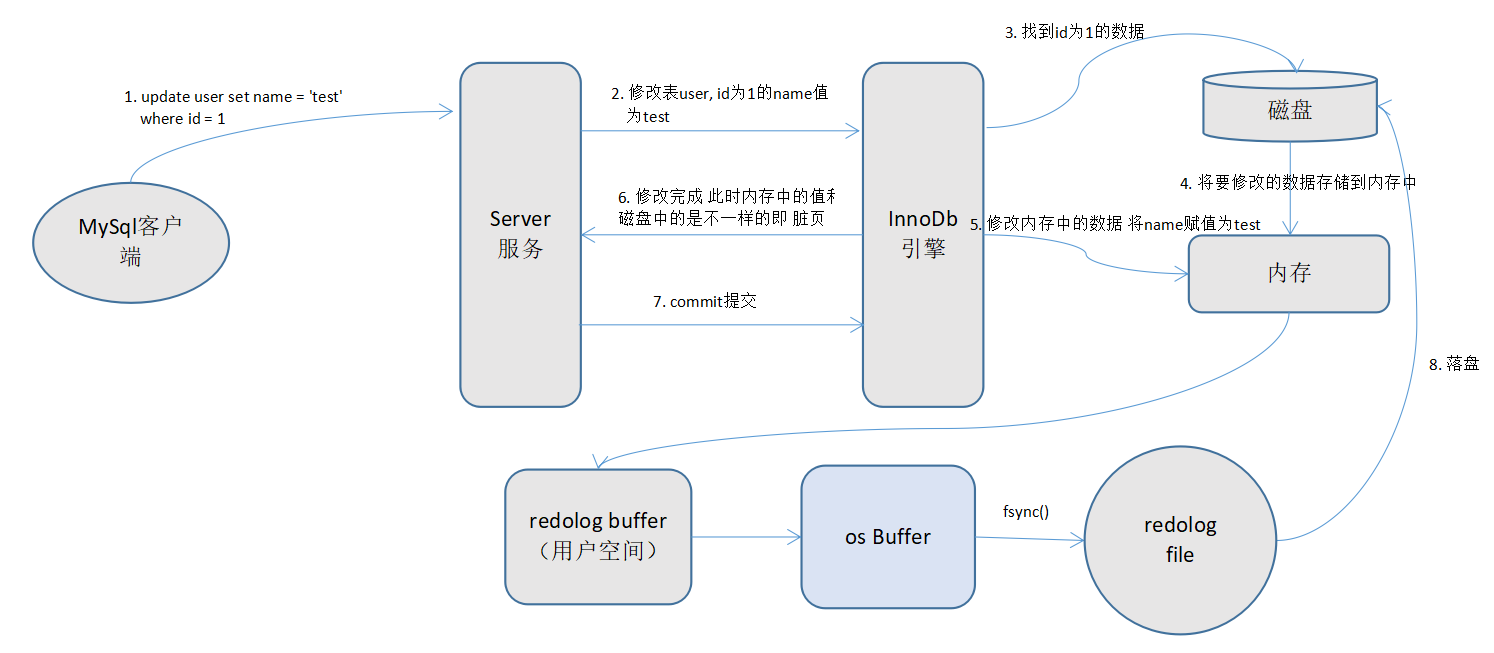
checkpoint就是负责落盘的，将数据存储到磁盘上并擦掉，由一个异步线程来执行的。

RedoLog的操作日志有两种状态，未提交时prepare状态，提交之后时commit状态。

RedoLog包含两部分，一个是内存中的日志缓冲（redo log buffer），另一个是磁盘上的日志文件( redo logfile)。缓冲主要是解决速度不一致引入的。另外缓冲和日志文件的交互需要经过os buffer（用户空间[user space]下的缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统内核空间[kernel space]缓冲区[OS Buffer]）

使用innodb\_flush\_log\_at\_trx\_commit配置来决定buffer->file的策略

1. 1（默认）时，事务提交都会将redolog buffer写入os buffer并调用fsync()刷到log file， 不会有数据丢失，但是性能差。
2. 0时，事务提交不讲redo log buffer写入os buffer，而是每秒执行dolog buffer->os buffer->dolog file， 系统崩溃，会丢失1钟数据。
3. 2时，每次提交都会写道os buffer，然后每秒调用fsync()将os buffer中的日志写入redo log file.



注意: binlog日志会优先于redolog日志产生

注意: redolog信息一定会在binlog中存在（逻辑一致），为了保持一致，它的提交流程如下

1 prepare阶段 -->  2 写binlog  --> 3 commit

1> 当在2之前崩溃时，重启恢复后发现没有commit，回滚。备份恢复：没有binlog 。一致

2> 当在3之前崩溃时，重启恢复发现虽没有commit，但满足prepare和binlog完整，所以重启后会自动commit。备份：有binlog. 一致

### 场景

#### 崩盘恢复

上面知道事务操作会先写入RedoLog中。RedoLog和数据页都会有各自的LSN（Log Sequence Number），根据LSN的值来判断恢复redolog的位置和大小，从而恢复已经提交的数据。

参照: https://www.cnblogs.com/f-ck-need-u/archive/2018/05/08/9010872.html

## Undo log

1. 逻辑日志，根据每行记录进行记录，比如一条 INSERT 语句，对应一条DELETE 的 undo log ，对于每个 UPDATE 语句，对应一条相反的 UPDATE 的 undo log ，这样在发生错误时，就能回滚到事务之前的数据状态。

数据库事务四大特性中有一个是 原子性 ，具体来说就是 原子性是指对数据库的一系列操作，要么全部成功，要么全部失败，不可能出现部分成功的情况。

实际上， 原子性 底层就是通过 undo log 实现的。undo log主要记录了数据的逻辑变化，同时， undo log 也是 MVCC(多版本并发控制)实现的关键。

## 事务

> 事务是数据库管理系统执行过程中的一个逻辑单位(把多件事情当一件事情做)

> 数据库底层的事务和回滚是通过bin log(归档日志)和redo log(重做日志)来实现的

> mysql默认的事务隔离级别是REPEATABLE-READ(可重复读)

> 基本属性(ACID)

原子性(Atomicity): 要么全部成功, 要么全失败

一致性(Consistency): 事务必须是数据库从一个一致性状态变到另一个一致性状态

隔离性(isolation): 一个事务内部的操作及使用的数据对并发的其他事务是隔离的

持久性(Durability): 事务结束后 对数据的改变时永久性的

> InnoDB支持事务 MyISAM不支持事务 事务支撑是在引擎层上实现的

### 简单操作

SELECT @@tx\_isolation # 查询当前事务的隔离级别

set session transaction isolation level read uncommitted #设置事务的隔离级别

start transaction #开启事务

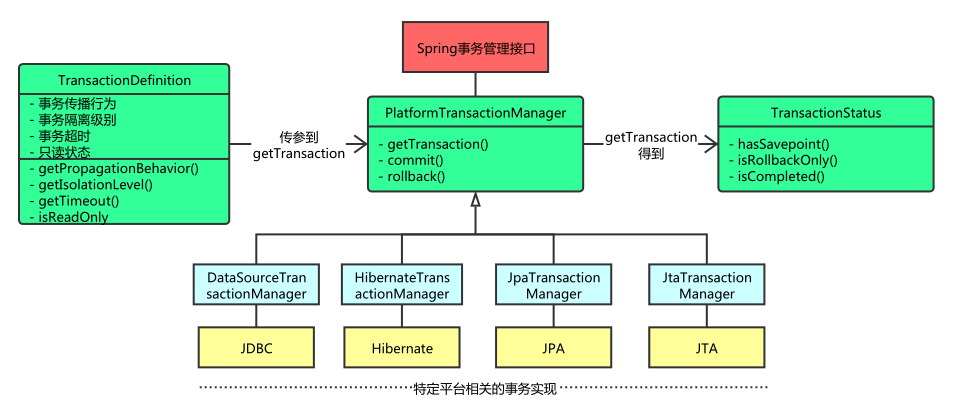
Rollback #回滚 commit #提交

### 事务的隔离级别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **事务** | **脏读** | **不可重复读** | **幻读** | **描述** |
| read uncommitted (未提交读) | √ | √ | √ |  |
| read committed (提交读) | × | √ | √ | 事务提交后才能被读取 |
| repeatable read( 可重复读) | × | × | √ | 事务执行期间 多次读取同一数据返回第一次快照 |
| serialization (可串行化) | × | × | × | 写加写锁 读加读锁, 读写冲突需前一个事务完成 |
| **脏读** | 事务2使用事务1未提交的数据 | | | |
| **不可重复读** | 事务1读取数据(未提交), 事务2对此数据进行修改提交, 事务1再次读取该数据 | | | |
| **幻读** | 事务1读取数据的条数, 事务2插入一条数据到该范围, 事务1再次读取数据的条数 | | | |

附件: <https://blog.csdn.net/qq_35433593/article/details/86094028>

### Spring Boot中事务的实现



spring不直接管理事务, 而是提供了一个PlaformTransactionManager接口, 由各平台来实现事务管理。

#### 事务的属性定义

事务的隔离级别: 事务执行期间可否被其它事务访问的配置

事务超时: 事务必须在规定时间内完成, 否则回滚

回滚规则: 指定那种情况下进行回滚, 默认是RuntimeException和Error, 可指定异常

事务是否只读: 确保读取多条数据的一致性(确保数据整体的一致性, 从开始时间a开始, a之后被提交的数据不可见)

事务的传播行为: 当一个事务方法被另方法调用时的传播行为。

方法A调用事务方法B, B是否继续在A的事务中执行, 取决于B事务的传播行为。

|  |  |
| --- | --- |
| **事务的传播行为** | **方法A 调用 事务方法B 时 事务的传播行为** |
| required | 继续使用A的事务, 若A没有事务, 则B自己创建一个 |
| supports | 继续使用A的事务, 若A没有事务, 则B也不使用事务 |
| mandatory | A必须有事务, B使用A的事务, 若A没有则异常 |
| required\_new | B总使用自己的事, A(事务方法时)和B是两个独立的事务, A挂起, B完成自己的事务, A继续 |
| not\_supported | A无论有没有事务 B都不会有事务 即B的运行是独立的 |
| never | A不能时事务方法, 否则抛异常 且B本身也不有事务行为 |
| nested | B总使用自己的事务, A(事务方法时)的事务嵌套B的事务, B随A的提交提交, 回滚而回滚 |

#### 使用事务

1. 启动类上添加@EnableTransactionManagement来开启事务(可通过其它配置使用不同的事务管理器)
2. 在service方法上添加@Transactional来使用事务(指定事务的属性)

该注解以下条件可能会失效 即不回滚

> 用于非public方法上, aop反射调用的时候会检查是不是public修饰的

> propagation的配置从而导致事务不会发生回滚

> 异常未捕获 跑出其它异常导致未回滚

> **同一个类中方法调用(A调用B A方法没有事务 B有事务 则B方法的事务失效)**

> 引擎不支持(很少有不支持的)

\* dbop中有demo可参照

## MVCC

### 简介

Mutil-Version Concurrenty Control多版本并发控制, 一般用在数据库管理系统中, 实现对数据库的并发访问.

InnoDb引擎的RC, RR这两个隔离级别使用了MVCC(乐观锁的实现)

### MVCC的使用

#### 版本链

索引表中聚集索引会有两个隐藏的列, trx\_id事务id, roll\_pointer上一个版本记录地址.

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **data** | **trx\_id** | **Roll\_point** |
| 主键索引(聚集索引) | 数据信息 | 事务ID | 上一个版本地址 |

#### RC RR中的使用

当进行读取的时候会取得一个ReadView 该ReadView中记录着未提交事务的版本连信息

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **操作** | **id** | **data** | **trx\_id** | **Roll\_point** |
| 修改(未提交) | 1 | 小明3 | 40 | 上一个版本地址  上一个版本地址 |
| 修改(未提交) | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改(已提交) | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |
| 添加(已提交) | 1 | 小明 | 10 | 上一个版本地址 |

##### RC中

1. 取数据时, 会取得当前的ReadView

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改(未提交) | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改(未提交) | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

1. 当前访问事务id如果是40则无法访问, 如果是10不存在版本链中, 则可以访问, 如果是20存在版本链中不可访问, 如果是25不存在版本链中, 可以访问。
2. 读取数据时会从当前事务id向下取, 进行对比, 取到一个可以访问的版本的数据信息。

##### RR中

1. 第一次取得的readview是

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改(未提交) | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改(已提交) | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

1. 第二次读取数据时仍会使用开始取得的readview, 而RC中第二次读取数据时可能取到一个新的readview

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修改(未提交) | 1 | 小明3 | 40 | 上一个版本地址  上一个版本地址 |
| 修改(已提交) | 1 | 小明2 | 30 | 上一个版本地址  上一个版本地址 |
| 修改(已提交) | 1 | 小明1 | 20 | 上一个版本地址  上一个版本地址 |

所以对于RC来说第一次读的是事务id是20的版本数据, 第二次读取的是事务为30的版本数据. RR会使用一个readview所以不存在不可重复读的问题.

## InnoDb和MySAM索引的比较

<https://blog.csdn.net/qq_35642036/article/details/82820178>

ALTER TABLE table\_name ROW\_FORMAT = DEFAULT

## Row Format

行格式规定了数据表中的每行数据在物理上是如何存储的, 即规定了每行数据的物理组织结构。不同的行格式, 是可以影响数据操作性能的。

### 操作

查看当前表使用的行格式

SHOW TABLE STATUS LIKE 'table\_name'

设置文件存储格式

set global innodb\_file\_format=Barracuda

修改表的行格式

ALTER TABLE table\_name ROW\_FORMAT = DEFAULT

### 描述

Antelope

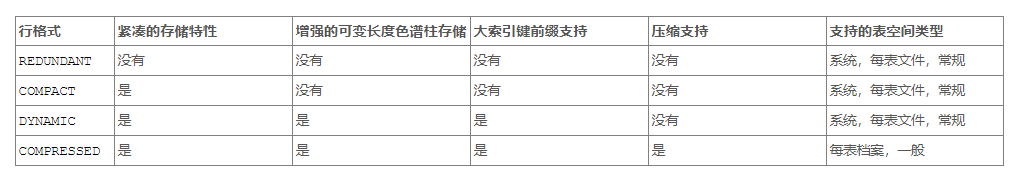
先前的未命名的原始的InnoDb文件格式. MySql5.6默认文件格式, 不支持Barracuda文件格式, 它的行格式有一下两种:

1. Compact
2. Redundant

Barracuda

新文件格式, 支持InnoDb的所有行格式, 包括新的行格式, 5.7.9之后默认是dynamic

1. Redundant
2. Compact
3. Dynamic
4. Compressed



附件: https://dev.mysql.com/doc/refman/8.0/en/innodb-row-format.html

附件: https://www.pianshen.com/article/80641156962/

## 分库, 分表

### 场景

#### IO瓶颈

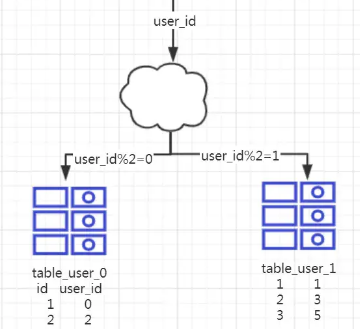
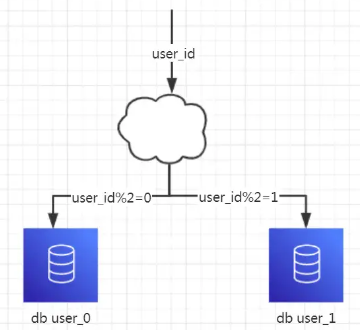
1. 磁盘ID瓶颈, 分库, 垂直分表
2. 网络IO瓶颈, 分库

#### CPU瓶颈

1. Sql查询耗费cpu, sql优化
2. 表单数据量大, 水平分表

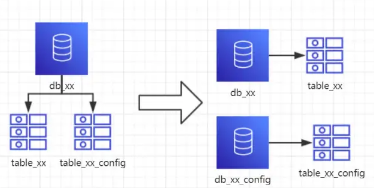
### 水平分库, 水平分表

1. 水平分库是按照一定的策略(hash, range)将一个库中的数据拆分到多个库中. 库结构相同, 库中的数据不同, 没有交集, 并集是全部数据. 场景: 系统并发量巨大.
2. 水平分表是按照一定的策略(hash, range)将一个表中的数据拆分到多个表中. 表结构相同, 表中的数据不同, 没有交集, 并集是全部数据. 场景: 系统并发量不是很大, 但是表中的数据量太大, 查询变慢.

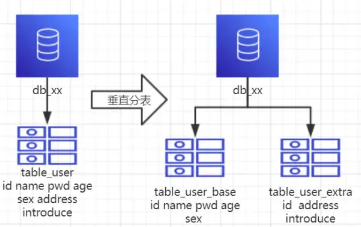


### 垂直分库, 垂直分表

1. 垂直分库, 以表为依据, 按照不同的业务, 不同的表放到不同的库中, 并集是全量数据. 场景: 业务的绝对并发量大, 且可以抽象出独立的业务模块。



1. 垂直分表, 以字段为依据, 按照字段的活跃性, 将表中的字段拆分到不同的表中(主表和扩展表), 每个表的字段至少有一列交集, 一般是主键用来关联表。并集是全量数据. 场景: 并发量不是很高, 但是表中的字段超多, 热点数据和非热点数据在一块, 查询产生大量的随机IO, 产生IO瓶颈. (建议: 不要使用join 最好是在service层做关联)



### 缺点

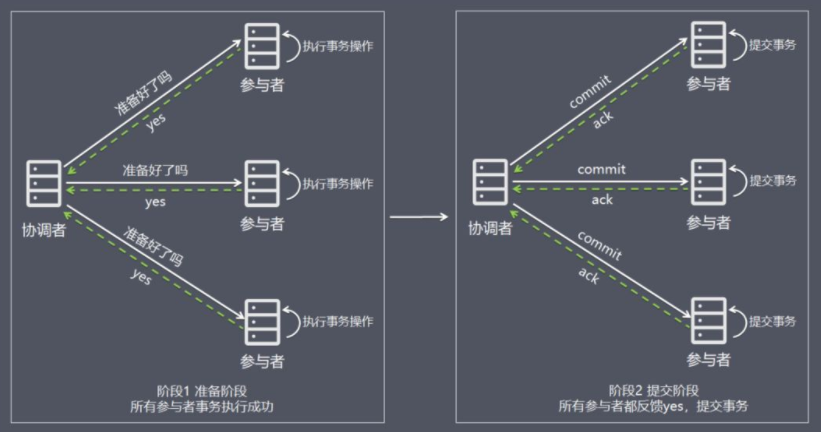
#### 分布式事务和最终一致性

当更新内容同时存在于不同的库中时, 就会有分布式事务的问题. 通常使用"XA"协议即"两阶段式提交"来解决. 但是分布式事务都会式最终的提交延后, 间接延长了事务的执行时间, 导致事务在共享资源的访问冲突和死锁概率增加. 数据库节点越多, 那么这种趋势也会越严重. 成为系统在数据库层面上水平扩展的枷锁.

系统不需要实时的强一致性, 允许在一段时间内达到最终的一致即可. 可以使用事务补偿等方式, 达到最终一致性

##### 4.1.1 **X/Open XA 事务机制 -** 两段式提交 2PC

事务的发起者叫协调者, 事物的执行者叫参与者. 由协调者来控制参与者进行统一行为。



**阶段 1：准备阶段**

> 协调者向所有参与者发送事务内容，询问是否可以提交事务，并等待所有参与者答复。

> 各参与者执行事务操作，将 undo 和 redo 信息记入事务日志中（但不提交事务）。

> 如参与者执行成功，给协调者反馈 yes，即可以提交；如执行失败，给协调者反馈 no，即不可提交。

**阶段 2：提交阶段**

> 如果协调者收到了参与者的失败消息或者超时, 直接给每个参与者发送回滚(rollback)消息；否则, 发送提交(commit)消息。

> 当所有参与者均反馈 yes, 协调者向所有参与者发出正式提交事务的请求(即 commit 请求), 参与者执行 commit 请求各参与者向协调者反馈 ack(应答)完成的消息。 协调者收到所有参与者反馈的 ack 消息后, 即完成事务提交。

> 当任何阶段 1 一个参与者反馈 no, 中断事务, 协调者向所有参与者发出回滚请求(即 rollback 请求), 参与者使用阶段 1 中的 undo 信息执行回滚操作，并释放整个事务期间占用的资源。 各参与者向协调者反馈 ack 完成的消息。协调者收到所有参与者反馈的 ack 消息后, 即完成事务中断。

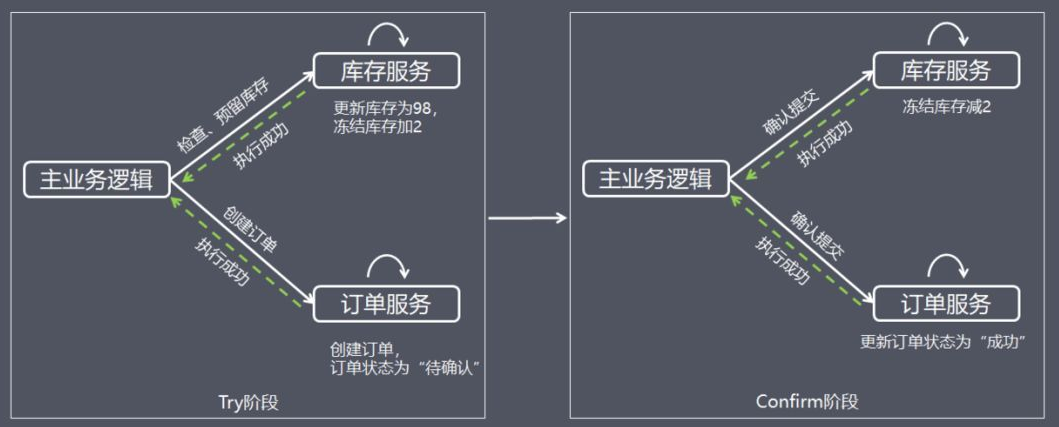
**缺点**

* 性能问题：所有参与者在事务提交阶段处于同步阻塞状态，占用系统资源，容易导致性能瓶颈。
* 可靠性问题：如果协调者存在单点故障问题，如果协调者出现故障，参与者将一直处于锁定状态。
* 数据一致性问题：在阶段 2 中，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

**附:** 三段式(3pc)提交是在2pc的基础上扩展, 将阶段2, 提交阶段分成了canCommit, preCommit, doCommit, 若协调者有问题, 那么doCommit, 超时后将继续提交.

##### 4.1.2 TCC (Try-confirm-cancel) 最终一致性机制

最终一致, TTC 可以理解为SQL事务中的Lock, Commit, Rollback.



**特点:**

性能提升, 根据具体的业务来控制资源的力度变小, 不会锁定整个资源

数据最终一致性, 基于Confirm和Cancel的等幂性, 保证是无罪总完成确认或者取消, 保证数据一致性

可靠性, 解决了XA协议的协调者单点故障问题, 由业务方发起并控制整个业务活动

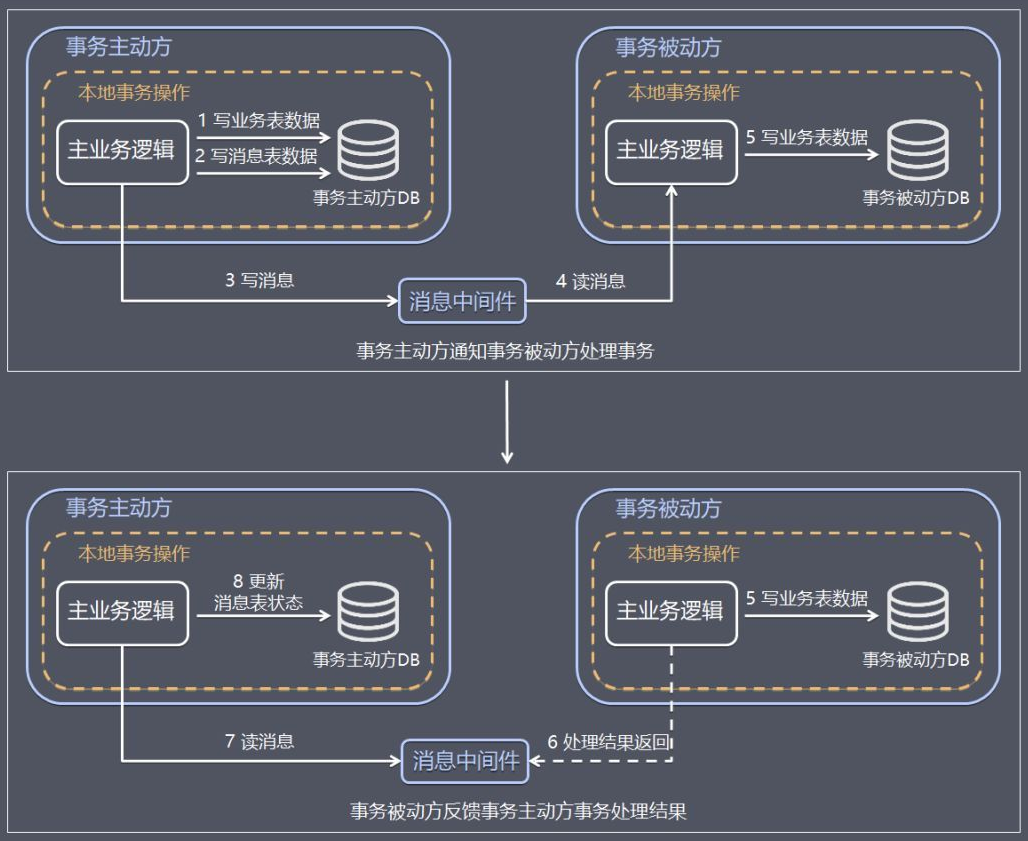
耦合度, 按具体业务来实现, 耦合度高, 提高开发成本

##### 4.1.3 本地消息表

通将分布式事务拆分成本地事务进行处理.

**特点:**

* 从应用设计开发的角度实现了消息数据的可靠性，消息数据的可靠性不依赖于消息中间件，弱化了对 MQ 中间件特性的依赖。
* 方案轻量，容易实现。
* 与具体的业务场景绑定，耦合性强，不可公用。
* 消息数据与业务数据同库，占用业务系统资源。
* 业务系统在使用关系型数据库的情况下，消息服务性能会受到关系型数据库并发性能的局限。



##### 4.1.4 MQ事务 最终一致性

**事务补偿**

未完待续...

附录: https://www.cnblogs.com/monkeyblog/p/10449363.html

#### Join问题

由于数据分布在不同的节点上, 原来需要使用join才能解决的问题, 现在变得很麻烦.

##### 全局表

全局表也可以看作是"数据字典表", 系统中所有模块都可能依赖的一些通常修改很少的常用表, 为了避免jion查询, 将该类表每个节点都保存一份.

##### 字段冗余

一种典型的反范式设计, 就是为了性能避免join查询, 在设计表的时候多保存了一些列, 适合依赖字段比较少的情况

##### 数据组装

在service 层进行两次查询, 然后根据关键字段组装数据

##### ER分片

也就是通过MyCat设置分片规则, 对表进行分片, 使得子表与与之关联的父表记录存放在同一个分片上

#### 分页, 排序, 函数问题

需要在每个分片上进行查询, 然后汇总, 计算

#### 4.4 全局主键避重问题

由于表中的数据同事存在于多个库中, 那么自增主键将无用武之地了.

UUID 可以作为主键, 但是因为其作为索引查询慢, 且无序性, 将导致索引位置频繁变动, 导致分页.

**SnowFlake 分布式Id子增算法**

### 分库分表时机

> 能部分就不分, 避免过度设计和过早优化, 分表分库将会是的业务的复杂的提升。

> 数据量过大, 已经影响业务时考虑

> 数据量增长速度过快时

## 分片, 分区, 分表, 分库

### 分片 sharding

MySQL5 以后提供了Sharding的能力, Sharding 是把数据库横向扩展(Scale Out)到多个物理节点上的一种有效的方式, 其主要目的是为突破单节点数据库服务器的 I/O 能力限制, 解决数据库扩展性问题. Shard这个词的意思是"碎片", 将整个数据库打碎的过程就叫做sharding, 可以翻译为分片。

### 分区

局限于单一节点, 将一张表分散到不同的物理块中, mysql5以后支持. 提高读写性能

其实就是 scale up 纵向扩展。

### 分表

应用层面上垂直分表, 水平分表. 对代码不透明. 物理层面上代理分表, 对代码透明. 分表就是把一个表变成多个表. 提高并发性能

### 分库

分库就是把单节点分成多节点, GA高可用.

## MyCat

# 缓存

缓存，就是将程序或系统经常要调用的对象（临时数据）存在内存中，以便其使用时可以快速调用，不必再去创建新的重复的实例。这样做可以减少系统的开销，提高效率。

使用缓存是为了减少和数据库的交互次数，提高执行效率。

经常查询的数据, 且数据不轻易发生变化适合缓存。 若数据经常变化且数据的正确与否对最终的结果影响很大则不适用于缓存。

## 操作系统缓存

附录: https://blog.csdn.net/carspiriter/article/details/80355066

## MyBatis缓存机制

mybatis的一级缓存是SqlSession级别的缓存，一级缓存缓存的是对象，当SqlSession提交、关闭以及其他的更新数据库的操作发生后，一级缓存就会清空。二级缓存是SqlSessionFactory级别的缓存，同一个SqlSessionFactory产生的SqlSession都共享一个二级缓存，二级缓存中存储的是数据，当命中二级缓存时，通过存储的数据构造对象返回。查询数据的时候，查询的流程是二级缓存>一级缓存>数据库。

# Redis

## 简介

Redis是一个开源的使用 ANSI C语言编写，支持网络，可基于内存也可持久化的日志型，Key-Value数据库，并提供了多种语言的 API ,相比 Memcached 它支持存储的类型相对更多 (字符，哈希，集合，有序集合，列表等)，同时Redis是线程安全的。

优点

1. 纯内存操作
2. 单线程操作, 避免了频繁的上下文切换
3. 采用了非阻塞I/O多路复用机制.

缺点

1. 缓存和数据库双写一致性问题
2. 缓存雪崩问题
3. 缓存击穿问题
4. 缓存的并发竞争问题

## 数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 简介 | 特性 | 场景 |
| String(字符串) | 二进制安全 | 可以包含任何数据,比如jpg图片或者序列化的对象,一个键最大能存储512M | 分布式锁, 计数器, |
| Hash(字典) | 键值对集合,即编程语言中的Map类型 | 适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去) | 存储、读取、修改用户属性 |
| List(列表) | 链表(双向链表) | 增删快,提供了操作某一段元素的API | 1,最新消息排行等功能(比如朋友圈的时间线) 2,消息队列 |
| Set(集合) | 哈希表实现,元素不重复 | 1、添加、删除,查找的复杂度都是O(1) 2、为集合提供了求交集、并集、差集等操作 | 1、共同好友 2、利用唯一性,统计访问网站的所有独立ip 3、好友推荐时,根据tag求交集,大于某个阈值就可以推荐 |
| Sorted Set(有序集合) | 将Set中的元素增加一个权重参数score,元素按score有序排列 | 数据插入集合时,已经进行天然排序 | 1、排行榜 2、带权重的消息队列 |

## Spring Boot集成redis

### 连接池

客户端连接 Redis 使用的是 TCP协议，直连的方式每次需要建立 TCP连接，而连接池的方式是可以预先初始化好客户端连接，所以每次只需要从 连接池借用即可，而借用和归还操作是在本地进行的，只有少量的并发同步开销，远远小于新建TCP连接的开销。另外，直连的方式无法限制 redis客户端对象的个数，在极端情况下可能会造成连接泄漏，而连接池的形式可以有效的保护和控制资源的使用。

|  |  |  |
| --- | --- | --- |
| 名称 | 优点 | 缺点 |
| 直连 | 简单方便，适用于少量长期连接的场景 | 1. 存在每次新建/关闭TCP连接开销 2. 资源无法控制，极端情况下出现连接泄漏   3. Jedis对象线程不安全(Lettuce对象是线程安全的) |
| 连接池 | 1. 无需每次连接生成Jedis对象，降低开销   2. 使用连接池的形式保护和控制资源的使用 | 相对于直连，使用更加麻烦，尤其在资源的管理上需要很多参数来保证，一旦规划不合理也会出现问题 |

### 常用的客户端

|  |  |  |
| --- | --- | --- |
| 名称 | 原理 | 其它 |
| Jedis | 直连 | 1. 线程不安全 2. 和redis指令一一对应, redis操作比较全面。 3. 同步阻塞, 不支持异步 |
| Lettuce | 基于Netty事件驱动通讯 | 1. 线程安全 2. 异步和响应使用，支持集群，Sentinel，管道和编码器 |
| Redisson | 基于Netty事件驱动通讯 | 1. 线程安全 2. 异步 3. 实现了分布式和可扩展的Java数据结构, 多用于分布式微服务 4. 不支持字符串操作，不支持排序、事务、管道、分区等Redis特性 |

\* jedis时springboot 1.5使用的, 若要线程安全需要为每个jedis实例增加物理连接, 不然多线程下不安全.

\* lettuce: https://www.cnblogs.com/throwable/p/11601538.html

\* 总体来说jedis淘汰了, lettuce还行, redisson 支持很多高级特性(高大上)

### 序列化

RedisTemplate默认使用的是JdkSerializationRedisSerializer，StringRedisTemplate默认使用的是StringRedisSerializer

**自带的序列化器**

1. GenericToStringSerializer
2. Jackson2JsonRedisSerializer
3. JacksonJsonRedisSerializer
4. JdkSerializationRedisSerializer
5. OxmSerializer
6. StringRedisSerializer。

Spring boot 缓存

<https://www.jianshu.com/p/e9b40acb2993>

## String

### 介绍

底层C语言中String用char[]数组表示，源码中用SDS(simple dynamic string)封装char[]，redis存储最小单元，一个SDS最大可以存储512M信息。

struct sdshdr {

unsigned int len; // 标记char[]的长度

unsigned int free; // 标记char[]中未使用元素的个数, 预分配用到

char buf[]; // 实际存放元素的地方

}

Redis底层对SDS有如下的优化:

1. SDS**修改后需要扩容**的话，系统会多分配空间来进行**空间预分配**

**\* 若SDS进行修改后，SDS的长度len小于1MB（SDS\_MAX\_PREALLOC），那么程序分配和len属性大小同样的未使用空间，这时SDS len = free，这样可以减少字符串修改时内存重分配次数，加快了程序执行速度。**

**\* 若SDS进行修改后的长度len大于等于1MB，那么程序会分配1MB的未使用空间。**

1. SDS是**惰性释放空间**的

**\* 若字符串缩短，不立即进行重分配来释放空间，而是将释放出来的字节转换成为空闲字节，同时更新free属性的大小**

**\* 可以避免缩短字符串带来的内存重分配操作，换可能为增长带来了缓存空间**

### 使用命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| SET key value [EX seconds] [PX milliseconds] [NX|XX] | 设置一个key value, 分布式锁 |
| setNx key value | 只有在 key 不存在时设置 key 的值 |
| Get key |  |
| Getrange key start end |  |
| Getset key value | 将给定key的值设置为value并返回旧的值 |
| Mget key1 [key2...] | 获取所有一个或多个给定key的值 |
| Setbit ket offset value | 对key所存储的字符串值，设置或者清除指定偏移量上的位(bit) |
| Settex key seconds value | 将值关联到key，并将key的过期时间设置为seconds（以秒为单位） |
| Setrange key offset value | 用value参数覆写给定key所存储的字符串值，从偏移量offset开始 |
| Strlen key | 返回key所存储字符串的长度 |
| [mset key value [key value ...]](https://www.runoob.com/redis/strings-mset.html) | 同时设置一个或多个 key-value 对。 |
| [msetnx key value [key value ...]](https://www.runoob.com/redis/strings-msetnx.html) | 同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。 |
| [psetex key milliseconds value](https://www.runoob.com/redis/strings-psetex.html) | 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。 |
| [incr key](https://www.runoob.com/redis/strings-incr.html) | 将 key 中储存的数字值增一。 |
| [incrby key increment](https://www.runoob.com/redis/strings-incrby.html) | 将 key 所储存的值加上给定的增量值（increment） 。 |
| [incrbyfloat key increment](https://www.runoob.com/redis/strings-incrbyfloat.html) | 将 key 所储存的值加上给定的浮点增量值（increment） 。 |
| [decr key](https://www.runoob.com/redis/strings-decr.html) | 将 key 中储存的数字值减一。 |
| [decrby key decrement](https://www.runoob.com/redis/strings-decrby.html) | key 所储存的值减去给定的减量值（decrement） 。 |
| [append key value](https://www.runoob.com/redis/strings-append.html) | 如果 key 已经存在并且是一个字符串， APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。 |

**set命令 redis2.6.12**

> EX secon: 设置键的过期时间为second 秒。 SET key value EX second效果等同于SETEX key second value

> PX millisecond: 设置键的过期时间millisecon毫。SET key value PX millisecon效果等同于PSETEX key millisecond value

> NX: 只在键不存在时，才对键进行设置操作。SET key value NX效果等同于SETNX key value。

> XX: 只在键已经存在时，才对键进行设置操作。

## Zset

### 简介

Redis 中的有序集合, 元素是string类型且不能重复, 集合的最大成员数是232 - 1即42多亿, 每个元素关联一个double类型的分数, double可以重复且用来进行排序。

Zset 当采用ziplist结构式crud的时间复杂度是O(1), 当采用skiplist时时间复杂度为O(log(n))。

### 使用命令

|  |  |
| --- | --- |
| 命令 | 说明 |
| Zadd(key, score, member) | 添加元素, 已存在会更新score即顺序 |
| Zrem(key, member) | 删除元素 |
| Zincrby(key, ncrement, member) | key存在score加ncrement, 否子添加score=ncrement |
| Zrank(key, member) | 返回key的zset从小到大, 没有返回nil |
| Zrevrank(key, member) | 返回key的zset从大到小, 没有返回nil |
| Zrange(key, start, end) | 返回key的zset从start到end 从小到大 |
| Zrevrange(key, start, end) | 返回key的zset从start到end 从大到小 |
| Zrangebyscore(key, min, max) | 返回min<=score<=max的zset元素 |
| Zcard(key) | 返回key的基数 |
| Zscore(key, element) | 返回它的score |
| Zremrangebyrank(key, min, max) | 删除min<=index<=max的zset元素 |
| Zremrangebyscore(key, min, max) | 删除min<=score<=max的zset元素 |

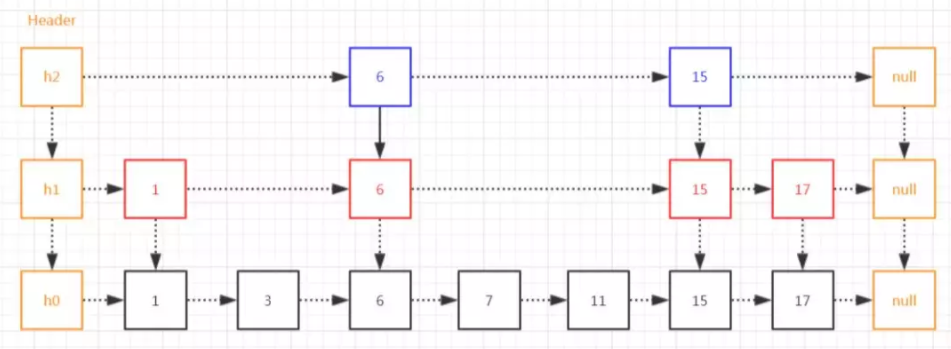
### 底层结构

Zset底层实现有两种方式

1. 压缩列表 ziplist
2. map和跳跃表skiplist, map保存score和member的意义映射。

#### 跳表 skiplist

通过对有序链表添加多级索引的方式优化查询速度, 对于链表而言增删改的速度取决于查询速度。



如上通过向上取索引的方法, 先二级索引然后一级然后链表这样搜索, 增加了查询速度(空间换时间)。

\* 概述

(1) 跳表是可以实现二分查找的有序链表

(2) 每个元素插入时随机生成它的level, redis中是这样的

(3) 最低层包含所有的元素

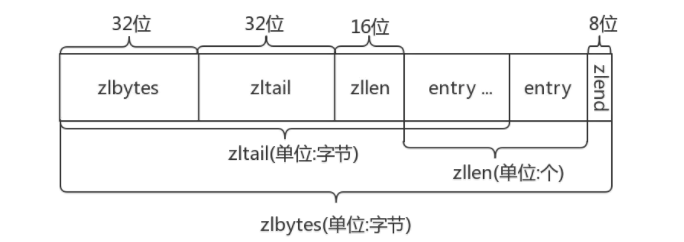
(4) 如果一个元素出现在level(x)，那么它肯定出现在x以下的level中

(5) 每个索引节点包含两个指针，一个向下，一个向右

(6) 跳表查询、插入、删除的时间复杂度为O(log n)，与平衡二叉树接近

\* Redis采用跳跃表的原因是查询时间复杂度与平衡二叉树接近, 并且跳跃表易读, 范围查询优于二叉树(二叉树要从头结点开始)。 作者说, 我就是觉得跳跃表简单好用。

#### 压缩列表 ziplist

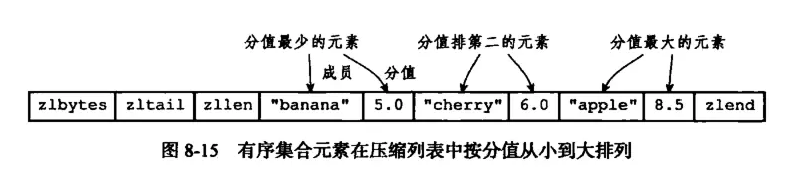
ziplist是由一系列特殊编码的连续内存块组成的顺序存储结构，类似于数组，ziplist在内存中是连续存储的，但是不同于数组，为了节省内存 ziplist的每个元素所占的内存大小可以不同（数组中叫元素，ziplist叫节点entry，下文都用“节点”），每个节点可以用来存储一个整数或者一个字符串。

Ziplist是为了节省空间而设计的, 每个entry中存储节点的偏移量从而找到上一个或下一个节点, 它可以使有序也可以无序, list和map底层是无序的, zset底层是有序的(从小到大)。

Zset存储满足一下条件采用ziplist结构:

1. 有序集合保存的元素数量少于128个。
2. 有序集合保存的所有元素的长度小于64字节。

\* 可以通过Redis配置文件zset-max-ziplist-entries 选项和 zset-max-ziplist-value 进行修改。



附录: https://www.jianshu.com/p/afaf78aaf615

### 使用场景

1. 延时队列: score采用时间搓, 根据时间搓判断进行消费达到延时队列的效果。
2. 排行榜: 例热门帖子, 帖子的ID当作member, 点击评论数作为score, 当score变化是进行更新, 取范围值即得到排行榜信息。
3. 限流: 用户Id作为member, score为用户访问的时间搓, 判端一段时间内用户的个数或一个用户的访问频率来限流。

## 缓存穿透

定义: 缓存穿透是指查询缓存层和存储层都不存在的数据, 导致每次查询都要去存储层查询从而失去缓存保护后端的意义。

原因: 自身代码或数据问题, 恶意攻击爬虫。

解决方法: 空值做缓存

## 缓存雪崩

定义: 缓存服务故障导致请求到存储层, 造成存储层死掉。

解决方法: 缓存高可用性, 隔离后端组件为后端限流并降级。

## 热点Key

定义: 当一个key是热点key

## 并发竞争

多个子系统去set同一个key

1. 不要求执行顺序, 分布式锁, 大家抢锁操作
2. 要求顺序, 使用mq串行执行或者保留每个子系统执行的时间搓, 来进行顺序控制。

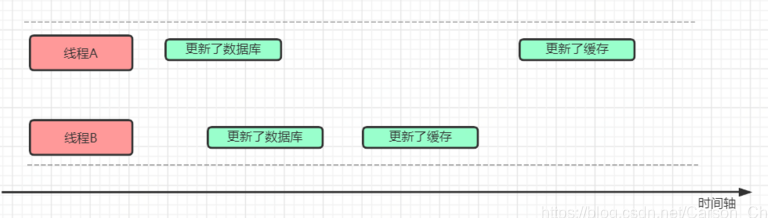
## 双写一致

对缓存数据进行修改时, 在高并发的场景下很容易发生数据不一致(缓存和数据库中数据不一致)问题。

### 问题

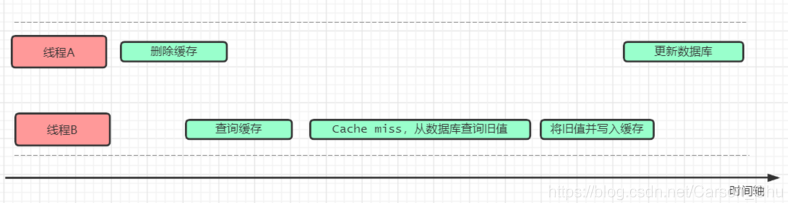
1. 读取时先缓存, 缓存没读数据库然后取出数据放入缓存中同时响应。
2. Cache Aside Pattern: 更新时先更新数据库, 然后删除缓存(为什么是删不是更新: 更新缓存可能有并发问题且可能浪费性能), 存在删除缓存失败或写操作比读操作快造成双写不一致的问题。

更新缓存存在问题:

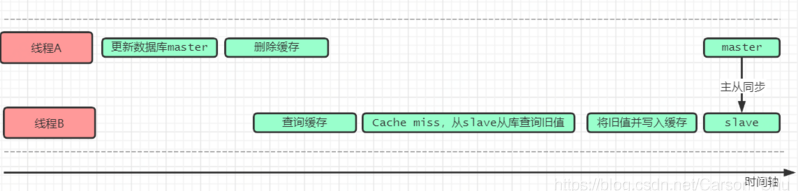


1. 更新时先删除缓存, 更新数据库。 存在更新数据库时别的线程提前取得旧数据更新到缓存, 造成双写不一致的情况。

高并发下:

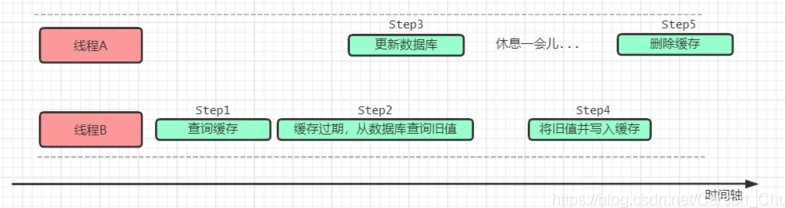


主从数据库时:



### 解决方案

1. 用一个队列进行串行化 不推荐
2. 延时双删, 即更新完删除, 过一会再删除一下(可以异步删除)



参照: https://blog.csdn.net/Carson\_Chu/article/details/108692533

1. 通过mq
2. 先删除缓存, 将更新操作放到内存mq中
3. 其它并发请求检测无缓存, 然后检测mq中有更新, 那么进行等待轮询
4. 若等待时间超过 降级处理返回旧数据
5. 若更新ok, 那么返回新的值

4. 设置redis的key的过期时间, 保证最终一致性。

# Java

## 基础

### 面向对象特征

封装 继承 多态

### JDK, JRE, JVM

#### JDK

Java Development ToolKit(Java开发工具包). JDK是整个JAVA的核心, 包括了Java运行环境(Java Runtime Envirnment)一堆Java工具(javac/java/jdb等)和Java基础的类库(即Java API 包括rt.jar)

常常用JDK来代指Java API, Java API是Java的应用程序接口, 其实就是前辈们写好的一些java Class, 包括一些重要的语言结构以及基本图形, 网络和文件I/O等等.

JDK包含JRE包含JVM

#### JRE

Java Runtime Enviromental(java运行时环境). 也就是我们说的JAVA平台, 所有的Java程序都要在JRE下才能运行. 包括JVM和JAVA核心类库和支持文件. 与JDK相比, 它不包含开发工具 编译器、调试器和其它工具

#### JVM

Java Virtual Mechinal(JAVA虚拟机), JVM是JRE的一部分, 它是一个虚构出来的计算机, 是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

JVM有自己完善的硬件架构，如处理器、堆栈、寄存器等，还具有相应的指令系统。JVM 的主要工作是解释自己的指令集(即字节码)并映射到本地的CPU的指令集或OS的系统调用。

Java语言是跨平台运行的，其实就是不同的操作系统，使用不同的JVM映射规则，让其与操作系统无关，完成了跨平台性。

JVM 对上层的 Java 源文件是不关心的，它关注的只是由源文件生成的类文件(class file), 类文件的组成包括 JVM 指令集，符号表以及一些补助信息

### 数组的copy

System.arraycopy： 指定数组, 指定副本进行数据copy. 底层使用c语言

Arrays.copy： 里面使用的是System.arraycopy, 它返回一个新的数组. arraycopy没有返回值, 必须传入副本.

### 不常用关键字

**Const**: 关键字保留, 但是不允许使用.

**Native**: 使用native关键字说明这个方法是原生函数, 也就是这个方法是用C/C++语言实现的, 并且被编译成了DLL, 由java去调用.

**Strictfp**: 修饰类, 接口, 方法. 修饰的作用域内所有浮点运算都是精确的, 符合IEEE-754的规范.

**Volatitle**: 使变量具有可见性, volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。

**Transient**: 被修饰的关键字不可被序列化. 原理: 生命周期只在内存中. 静态变量不会被序列化. 瞬态变量也不可以.

Serializable: 实现该接口 添加uuid.

Exteranlizable: 实现该接口的writeExternal和readExternal效率高于Serializable可以决定哪些属性需要序列化(即使是transient修饰的), 但是对大量对象, 或者重复对象, 则效率低.

**Goto**: 关键字保留, 但是不允许使用.

### Try catch finally

1、不管有没有异常，finally中的代码都会执行  
2、当try、catch中有return时，finally中的代码依然会继续执行  
3、finally是在return后面的表达式运算之后执行的，此时并没有返回运算之后的值，而是把值保存起来，不管finally对该值做任何的改变，返回的值都不会改变，依然返回保存起来的值。也就是说方法的返回值是在finally运算之前就确定了的。  
4、如果return的数据是引用数据类型，而在finally中对该引用数据类型的属性值的改变起作用，try中的return语句返回的就是在finally中改变后的该属性的值。  
5、finally代码中最好不要包含return，程序会提前退出，也就是说返回的值不是try或catch中的值

### Static

1、父类中的静态方法可以被继承、但不能被子类重写。

2、如果在子类中写一个和父类中一样的静态方法，那么该静态方法由该子类特有，两者不构成重写关系。

3、静态内部类才可以声明静态方法。

4、静态内部类不可以直接访问外围类的非静态数据，而非静态内部类可以直接访问外围类的数据，包括私有数据。

5、静态方法不可以访问非静态变量。

### Final

1、修饰类表示不允许被继承。

2、修饰方法表示不允许被子类重写，但是可以被子类继承，不能修饰构造方法。

3、修饰变量表示不允许被修改

a）方法内部的局部变量，使用前被赋值即可（只能赋值一次），没有必要非得初始化。

b）类中的成员变量（如果没有在定义时候初始化，那么只能在构造代码块中或者构造方法中赋值）

c）基本数据类型的变量（初始化赋值之后不能更改）

d）引用数据类型的变量（初始化之后不能再指向另外一个对象，但对象的内容是可以变的）

### 变量

**整形:** 0123 八进制 0xadf 十六进制 0b1010 二进制 1\_2\_323和1\_\_2\_\_32视觉分组

**字符:** '/n' '/'' '/"' '/ddd' '/uxxxx'

**整形转换成更小的类型:** 对更小的类型进行取模, 然后取余数。

**浮点变整形:** 对小数点及后面内容进行截断。

**类型提升规则:**

> byte, char, short 作为操作数会被提升为int

> 如果一个操作数是long, float则整个表达式提升为long, float

> 如果任何一个操作数为double则结果为double类型.

### String

1. 字符串是final类型的, 即不可变的
2. 创建字符串时, 首先会从字符串常量(字面量)池中寻找, 找到返回引用, 找不到则放入常量池, 然后返回引用。
3. 使用new创建字符串时, 总会创建一个新的字符串。
4. 使用intern可以将new出来的字符串存入常量池中(存在返回引用, 不存在存入)
5. 注, 字符串常量池存放的是对象的引用不是对象, 对象在堆中。
6. 字符串+ += 拼接会在编译阶段创建一个StringBuffer对象来拼接，所有不会产生多余的字符串对象。

### 其它

**Break:** 通常用来跳出循环, 设计初衷是具有go的作用, 调到标记位, 该标记代码块必须包含break否则编译失败.

**Continue：** 跳出本次循环, 也可以指定标记调到包含continue的标记位.

**可变参数(...):** 在一定的条件下具有模糊性, 编译器也无法理解, 需要自己另辟蹊径.

例: a(int a, int... b)和a(int a)当方法调用是a()或者a(1)的话, 是具有模糊性的。

## 数据结构

### List集合

#### 迭代元素删除

1. 使用iterator对元素进行迭代, for each内部也是用的是iterator, next会判断预期值和操作值是否相等, 即不允许增加和删除操作, 在迭代的时候。 **只能使用它自己的iterator.remove来进行删除(非线程安全).**
2. 使用集合的size进行for循环遍历操作, 当删除时注意索引值的变动, 否则会出现IndexOutOfBoundsException异常.

### 阻塞队列

#### 概论

满足两个附加操作的队列即为阻塞队列. 附加操作: 在队列为空时获取元素的线程会等待元素变为非空, 当队列满时, 存储元素的线程会等待队列可用. 阻塞队列常用于生产者消费者模式.

#### 方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **方法/处理方式** | **抛出异常** | **返回特殊值** | **一直阻塞** | **超时退出** |
| IllegalStateException/NoSuchElementException | True/False/返回值/Null | 一直阻塞或者响应中断退出 | 超过时间就退出 |
| 插入方法 | add(e) | offer(e) | put(e) | offer(e, timr, unit) |
| 移除方法 | remove() | poll() | take() | poll(time, unit) |
| 检查方法 | element() | peek() | 不可用 | 不可用 |

boolean add(E e) : 将给定元素设置到队列中, 如果设置成功返回true, 否则抛出异常. 如果是往限定了长度的队列中设置值, 推荐使用offer()方法.

boolean offer(E e) : 将给定的元素设置到队列中, 如果设置成功返回true, 否则返回false. e的值不能为空, 否则抛出空指针异常.

void put(E e) throws InterruptedException : 将元素设置到队列中, 如果队列中没有多余的空间, 该方法会一直阻塞, 直到队列中有多余的空间.

boolean offer(E e, long timeout, TimeUnit unit)throws InterruptedException : 将给定元素在给定的时间内设置到队列中, 如果设置成功返回true, 否则返回false.

E take() throws InterruptedException : 从队列中获取值, 如果队列中没有值, 线程会一直阻塞, 直到队列中有值, 并且该方法取得了该值.

E poll(long timeout, TimeUnit unit) throws InterruptedException : 在给定的时间里, 从队列中获取值, 如果没有取到会抛出异常.

int remainingCapacity() : 获取队列中剩余的空间.

boolean remove(Object o) : 从队列中移除指定的值.

public boolean contains(Object o) : 判断队列中是否拥有该值.

int drainTo(Collection<? super E> c) : 将队列中值, 全部移除, 并发设置到给定的集合中.

int drainTo(Collection<? super E> c, int maxElements) : 指定最多数量限制将队列中值, 全部移除, 并发设置到给定的集合中.

#### 家庭成员

|  |  |  |  |
| --- | --- | --- | --- |
| **队列** | **有界性** | **锁** | **数据结构** |
| ArrayBlockingQueue | Bounded(有界) | 加锁 | ArrayList |
| 支持公平锁和非公平锁, 按照FIFO的原则对元素进行排序 | | | |
| LinkedBlockingQueue | Optionally-bounded | 加锁 | linkedList |
| 默认长度为Integer.MAX\_VALUE | | | |
| PriorityBlockingQueue | unbounded | 加锁 | Heap |
| 支持优先级排序, 默认自然序进行排序, 可以实现compareTo()接口, 不能保证同优先级元素的顺序. | | | |
| DelayQueue | unbounded | 加锁 | heap |
| 一个实现PriorityBlockingQueue实现延迟获取的无界队列, 在创建元素时, 可以指定多久才能从队列中获取当前元素. 只有延时期满后才能从队列中获取元素. 例: TimeQueue是使用DelayQueue实现的 | | | |
| SynchronousQueue | bounded | 加锁 | 无 |
| 一个不存储元素的阻塞队列, 每一个put操作必须等待take操作, 否则不能添加元素. 支持公平锁和非公平锁 | | | |
| LinkedTransferQueue | unbounded | 加锁 | Heap |
| 由链表结构组成的无界阻塞队列, 相当于其它队列, LinkedTransferQueue队列多了transfer和tryTransfer方法 | | | |
| LinkedBlockingDeque | unbounded | 无锁 | Heap |
| 由链表结构组成的双向阻塞队列, 队列头部和尾部都可以添加和移除元素, 多线程并发时, 可以将锁的竞争最多降到一半 | | | |

附录: https://www.cnblogs.com/bjxq-cs88/p/9759571.html

### HashMap

特点

1. 初始值16, 不大不小.
2. 加载因子0.75 计算得来的, 可以使得值均匀分布.
3. 容量会为8的倍数.
4. 链表长度大于8转换成红黑树(并且要求整个node节点的數量要大于64), 红黑树小于6转换成链表, TreeNode占用的空间比Node大。
5. 迭代删除和list一样
6. 添加时先比较hashCode的结果, 在比较equals的结果, String, Integer这些类型的hashCode被从写, 至于值有关.

https://blog.csdn.net/u013182960/article/details/80225454

### IdentityHashMap

1. 存储数据时, 使用的时对象的地址来分辨数据的. System.identityHashCode()
2. 允许key和value都为null

### ConcurrentHashMap

特点

1. HashMap并发可能发生死循环, HashTable安全但是效率低下.
2. HashTable只有一把锁, 而ConcurrentHashMap对数据分段, 每一段采用一把锁, 即锁的分段技术.
3. 使用Segment数组包含一段HashEntry
4. 1.8 抛弃了Segment采用CAS + synchronized
5. 可以再迭代的时候进行添加删除, 因为其是**弱一致性的**
6. Hashcode 采用高低16位亦或来加大低位的随机性, 并且适用hashCode & (len-1)来提升效率
7. 初始化时使用的是cas来进行安全, 其它线程使用yeild放弃执行资源。
8. Segment优化, 将头节点对象作为所对象, 进行分片锁(该所只作用于它下面的链表或者红黑树)
9. 数组的扩展为什么必须是2的n次幂, 因为这个对于计算hash code有影响, 会导致计算的重复率提升
10. key和value都不能是null
11. 数组扩容数据迁移, 多个线程领取任务(16个数据)的方式合作完成迁移, 即当迁移过程中, 修改节点的标记hashCode -1, 其它线程put是, 检测到是迁移中, 那么就帮助迁移.

#### HashSet

1. 不重复, 顺序没有保障, 元素可以为null, 但是只有一个, 非线程安全
2. 判断是否是一个值, 必须equals和hashCode**都**相等才行

#### LinkedHashSet

特点:

1. 初始化大小16, 加载因子0.75, 元素不重复, 有序(根据插入顺序)
2. 迭代顺序等于插入顺序, 双向链表保证

#### TreeSet

特点:

1. 确保集合元素处于排序状态, 可以定制排序

### 接口

**default方法**

在接口中被default修饰的方法为普通方法. 只能被接口的实现类的对象调用, 可以被重写.

> 该方法可以被子类继承使用

> 当子类实现两个接口, 且接口中都有default A 那么A必须被重写, 否则编译失败.

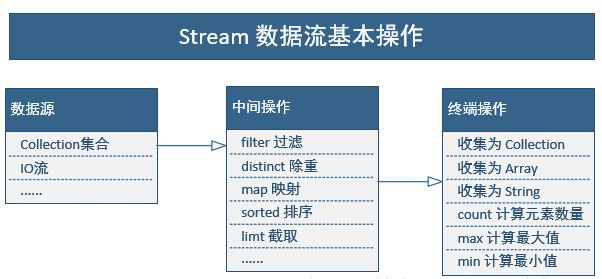
> 当子类继承一个类实现一个接口, 父类中有A, 接口中有default A那么实际继承类父类的A方法.

**static方法**

在接口中用static修饰的方法为接口的类方法, 只能被接口调用, 不能被实现类或者实现类的对象调用.

### Stream

每个流代表一个值序列, 流提供一系列常用的聚集操作.



#### 流的特点

1. 只能遍历一次
2. 采用内部迭代方式, 大数据量比外部迭代(Iterator)更高效
3. 相对于Collection的优点

无存储, 函数式风格(流的操作会产生一个结果, 不会修改源头), 惰性求值, 无上界, 代码简练

1. 迭代效率

传统的Iterator比Stream迭代效率高, 尤其是小数量的情况

在多核, 大数据量的情况下, parallel stream可以有比iterator更高的迭代处理效率.

### JVM底层

Class Loader类加载器

1. 定位和导入二进制的.class文件, 是否可以运行由Execution Engine决定

Execution Engine执行引擎

也叫做解释器, 负责解释命令, 交给操作系统执行.

#### 对象头

在HostSpot VM(Sun JDK和OpenJDK中所带的虚拟机, 也是目前使用范围最广的Java虚拟机)中对象头主要包含两部分, Mark Word(标记字段), Klass Pointer(类型指针)。

其中类型指针时对象指向它的类元数据的指针, 虚拟机通过这个指针来确定这个对象是那个类的实例, 标记字段是存储对象自身的运行时数据。

对象头一般是2个机器码(一个机器码4个字节), 如果是数组类型则需要3个, 其中一个记录数据长度。Mark Word会根据对象的状态来复用自己的存储空间达到空间效率(在极小的空间内存储尽量多的数据)。

下图是32位的, 64位和32位基本类似。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **锁状态** | **25bit** | | **4bit** | **1bit** | **2bit** |
| **23bit** | **2bit** | **是否是偏向锁** | **锁标志位** |
| 无状态锁 | 对象hashcode | | 对象分代年龄 | 0 | 01 |
| 轻量级锁 | 指向锁记录的指针 | | | | 00 |
| 重量级锁 | 指向重量级锁的指针 | | | | 10 |
| GC标记 | 空 | | | | 11 |
| 偏向锁 | 线程ID | Epoch | 对象分代年龄 | 1 | 01 |

#### 对象的引用

把对象分成四种类别, 从而使程序更能灵活(专解决内存操作不灵活问题)的控制对象的生命周期, 进行高效的垃圾回收.

##### 强引用

new出来的对象都属于强引用.

对象一直处于可触及状态, 垃圾回收器不会回收, 内存空间不足会抛出OutOfMemoryError. 常见的有集合中的对象, 这些对象就属于强引用, 一般推荐clear有利于垃圾回收.

##### 软引用

鸡肋的对象, 内存空间足够不会回收它, 内存空间不足会进行回收, 和SoftReference配合使用.

##### 弱引用

可有可无, 生命周期短暂, 一旦发现弱引用, 不管内存是否充足都会回收, 和WeakReference配合使用. 垃圾回收器优先级低, 一般也不好发现弱引用.

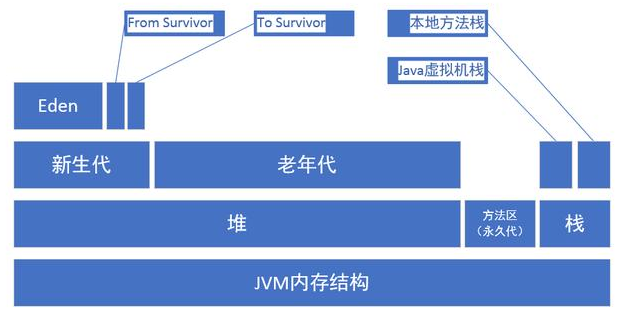
##### 虚引用

和没有引用一样, 任何时候都能被回收, 主要用来跟踪对象垃圾回收的活动, 和ReferenceQueue联合使用.



#### JVM内存结构

Jdk1.7及以前使用的是sun公司的HotSpot的JVM, 之后由于sun公司和BEA(JRockit JVM)公司都被oracle收购, jdk1.8采用两个原来公司jvm的精华形成了jdk1.8的JVM.



##### 堆(Heap)

1. 线程共享区域, 虚拟器启动时创建的, 垃圾回收的主要地方
2. New出来的对象都存在于堆中(基本上是, 不是非常绝对的)
3. 堆=年轻代 + 老年代, 年轻代= Eden + From Survivor + To Survivor, Form = To(大小上).

默认的: 年轻代:老年代=1:2, Eden:From:To=8:1:1

##### 方法区(Method Area)

1. 线程共享内存区域, 又称永久代(连续的堆空间), 默认64M(64位JVM默认85M)
2. 存储虚拟机加载的类信息, 常量, 静态变量
3. JDK8之后不再有永久带, 类元数据信息(metadata)移动到叫做metaspace的本地内存(Native memory), 不再是连续的堆空间.

##### 虚拟机栈(JVM Stack)

1. 线程私有, 生命周期和线程相同, 每个方法会创建一个栈帧, 方法的执行就是进栈出栈的过程
2. 栈帧有: 局部变量表, 操作数栈, 动态链接, 方法出口等

##### 本地方法栈(Native Stack)

与虚拟机栈相同, 不过是为虚拟机使用到的Native方法服务.

##### 程序计数器(PC Register)

1. 线程私有的
2. 记录当前线程执行程序的位置, 改变计数器的值来确定执行下一条指令, 比如循环、分支、方法跳转、异常处理, 线程恢复都是依赖程序计数器来完成。
3. Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。

##### 直接内存

jdk1.4引入NIO, 引入通道和缓冲区的IO方式, 可以调用Native直接在堆外分配内存, 不影响堆内存大小.

#### 垃圾回收

##### 回收算法

###### 标记清除算法

1. 通过根节点(GC Roots), 标记所有从根节点开始的对象. 通常的根节点为栈, 静态变量, 寄存器(寄存器是CPU内部用来存放数据的一些小型存储区域, 用来暂时存放参与运算的数据和运算结果).
2. 未被标记的对象即未被引用的对象, 在清除阶段清除所有未被标记的对象.

优点:

存活对象较多的情况, 适用于年老代。

缺点:

容易产生内存碎片, 有大对象时容易提前触发垃圾回收。 扫描整个空间两次(标记, 清除)

###### 复制算法

1. 通过根节点(GC Roots), 标记所以从根节点开始的对象.
2. 移动这些标记的对象到新的区域, 回收空着的区域

优点

存活对象少的情况比较高效, 适用于年轻代

缺点

扫描一次, 需要一块空间, 需要复制移动对象

###### 标记整理算法

标记清除算法的优化, 标记之后将存活的对象压缩成一块连续的空间, 清除释放两端的空间

###### 分代收集算法

将内存区域分为各个年代, 不同对象的生命周期是不一样的, 对不同的年代使用不同的算法, 可以提高回收效率。

1. 新生代(Yong Generation) 复制算法
2. 老年代(Tenured Generation) 标记整理法
3. 持久代(Permanet Generation) 1.8之后已删除

回收过程

1. 新new的对象一般会在Eden区
2. 当Eden放不下会复制存活对象到From中, 之后Eden全部回收掉
3. 当From满了, 那么会复制Eden+From的存活对象到To中, 然后回收掉Eden和From, 然后交换FROM和To的指向进行2步.
4. 2,3 进行循环对象会被复制很多次, 每复制一次对象的年龄就会+1, 默认情况下当复制了15次会进入年老代, 可以通过-XX:MaxTenuringThreshold来配置。
5. 当年老代满了, 要放不下的时候, 会发生一次FullGC(耗时严重)

回收类型

**Minor Gc**

对新生代进行回收, 不会影响到年老代, Minor GC非常频繁, 当对象创建年青带内存不足时触发。

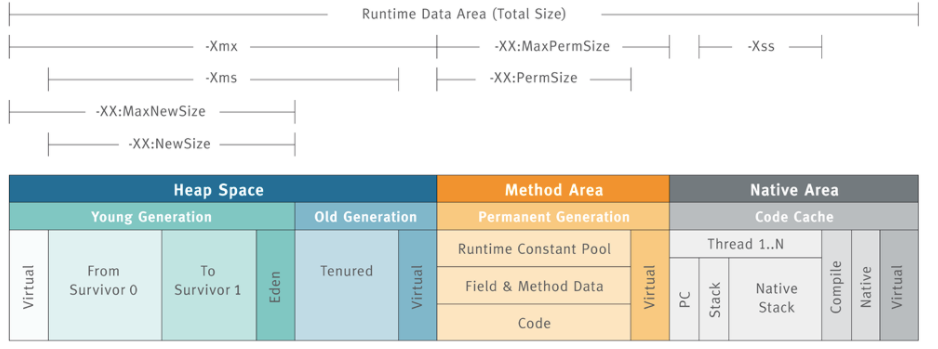
**Full GC (Major GC)**

对整个堆进行回收, 包括新生代和年老带. 导致Full GC的原因: 年老带被写满, 永久代被写满, System.gc() 显示调用, 上一次分配后堆区域分配策略动态变化.

##### 垃圾收集器

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 收集器名称 | 描述 | 优点 | 缺点 | 配置(打开) |
| 串行收集器 | 单线程处理所有垃圾回收 | 效率高, 适合单核, 多核小于100M比较好 | 对多核来说资源使用不到位, 只能应用于小型应用 | -XX:+UseSerialGC |
| 并行收集器 | 多线程并行处理垃圾回收 | 吞吐量高, 年青带是并行, 年老代还是串行需要设置 | 垃圾收集时系统响应时间加长 | -XX:+UseParallelGc  -XX:+UseParallelOldGc  -XX:ParallelGCThreads=<N> |
| 并发收集器 | 应用不停停止, 使用独立的垃圾回收线程(其实有很短暂的暂停) | 适合响应要求比较高的中, 大规模应用 | 需要预留20%空间进行浮动垃圾回收, 否则(Concurrent Mode Failure)回收失败将暂停程序进行回收 | -XX:+UserConcMarkSweepGc  -XX:CMSInitiatingOccupancyFraction=<N>指定剩余多少堆空间时进行回收 |

##### JVM配置参数



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 配置参数 | 默认值 | 推荐值 | 描述 | 例 |
| -XX:NewRatio | 1:2 |  | 新生代和老年代的比例 | -XX:NewRation=4 |
| -XX:SurvivorRatio | 8:1:1 |  | 新生代中空间的比例 | -XX:SurvivorRatio=4 |
| -Xss | 1M |  | 每个线程的堆栈大小 | -Xss128k |
| -XX:pretenureSizeThreshold | 0 |  | 大于该值创建对象会直接进入持久代中 |  |
| -XXMaxGCPauseMillis=<N> | 单位毫秒 | 最大垃圾回收暂停时间, 设置此值, 堆大小和相关参数会进行调整已达到此值, 可能会减少应用的吞吐量. | | |
| -XX:GCTimeRatio=<N> | 99 | 公式1/(1+N), 默认是1%的时间用来垃圾回收 | | |
| -Xmx |  |  | 堆最大值 | -Xmx3550m |
| -Xms |  |  | 堆最小值(最大最小相等可以避免JVM重新分配) | -Xms3550m |
| -Xmn | 堆的3/8 |  | 年轻代大小 | -Xmn2g |
| -XX:MaxPermSize | 64m |  | 持久代大小 | -XX:MaxPermSize=16m |
| -XX:+UseAdaptiveSizePolicy | 自动配置年轻代和响应Survivor的比例, 达到系统规定最低响应时间和收集频率等, 建议并行收集器是一直打开 | | | |
| -XX:CMSFullGCsBeforeCompaction | 并发收集器不压缩空间容易产生碎片, 降低运行效率, 设置多少次收集后对其压缩 | | | |
| -XX:+UserCMSCompactAtFullCollection | 打开对老年代的压缩, 可能影响性能, 但是可以消除碎片 | | | |
| -XX:+PrintGC | 垃圾回收统计信息 | | | |
| -XX:+PrintGCDetails | 垃圾回收统计信息 | | | |
| -XX:+PrintGCTimeStamps | 垃圾回收统计信息 | | | |

##### 增量收集和G1

由于传统的垃圾回收, 当处于应用及时性很高的场景下时, Full GC时带来的应用暂停会导致大量的请求堆积和请求失败, 为了达到实时性的要求所以诞生了增量收集.

G! 专为大型应用设计集成了分代回收和增量收集的优点。

#### 性能调优工具

##### jps

Java Virtual Machine Process Status Tool 基础工具, 主要用来输出JVM中运行的进程状态信息。Jdk1.5之后bin下的一个命令。

jps [options] [hostid]

如果不指定hostid 默认为当前主机或服务器

-q 不输出类名, jar名和传入main方法的参数

-m 输出传入main方法的参数

-l 输出main类或jar的全限名

-v 输出传入JVM的参数

##### jstack

jstack主要用来查看某个Java进程内的线程堆栈信息. JDK中自带的命令.

jstack [options] pid 常用

jstack [options] executable core

jstack [options] [server-id@]remote-hostname-or-IP

状态诺列:

Deadlock：死锁, 多个线程竞争同一个资源

Waiting on condition：等待资源或等待某个条件的发生

Waiting on monitor entry ：等待获取监视器

Blocked ：阻塞, 长时间等待未获取资源

Runnable 执行中

Suspended 暂停

Object.wait()或TIME\_WAITING 对象等待

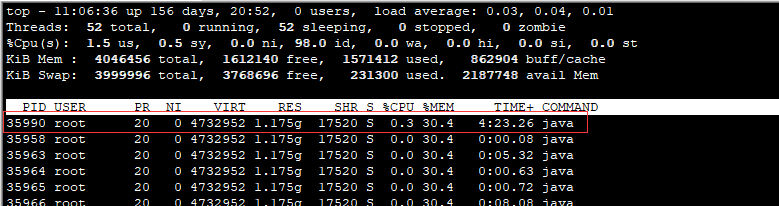
Parked 停止

例：

1. Jps -m 找到自己想要的运行中java项目的pid



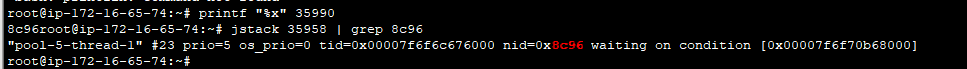
1. Top -Hp pid 选择该进程中 耗时最久的线程



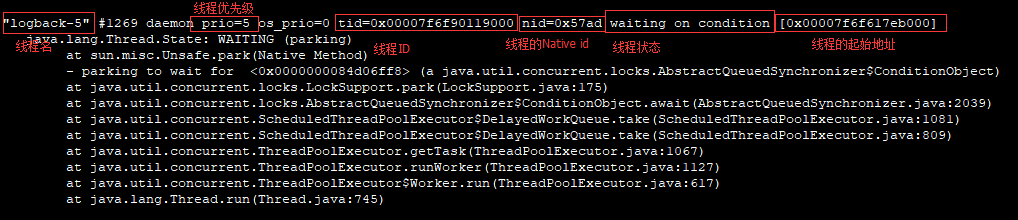
1. printf "%x" pid 将pid 转为16进制



1. Jstack pid | grep 十六进制子线程 找到原因



1. 例子



##### jmap和jhat

jmap(memory map)和jhat(java heap analysis tool), jmap导出堆内存所有对象及其数量并可生成文件, jhat来进行分析. jdk中命令.

jmap [options] pid 常用 链接正在运行的线程

jmap [options] executable core 链接到核心文件

jmap [options] [server-id@]remote-hostname-or-IP 连接到远程调试服务

常用命令

jmap -heap pid 查看堆内存占用情况

jmap -J-d64 -heap pid 64位的 查看堆内存占用情况

jmap -dump:live,format=b,file=myjmapfile.dump pid 输出head内容到文件

jmap -finalizerinfo pid 打印正在等待回收的信息

jmap -histo:live pid 打印堆中的每个class的实例信息 数量等

附录: https://www.cnblogs.com/sxdcgaq8080/p/11089664.html

##### Jstat

jvm统计监测工具

##### Hprof

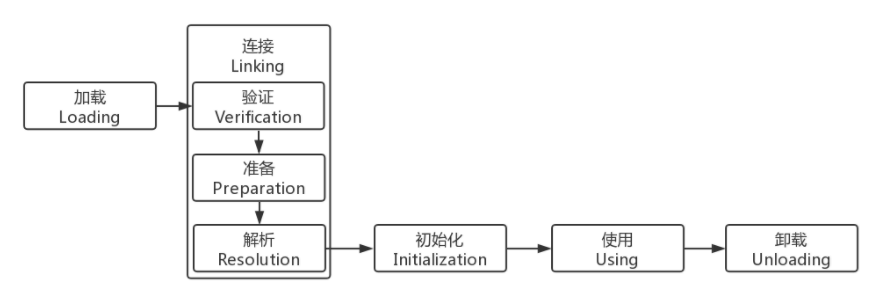
Hprof能够展现CPU使用率, 统计堆内存使用情况.

##### VisualVm

Visualvm用来分析内存使用情况, jdk自带的在bin下执行jvisualvm.exe

### 类加载机制

虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行验证、准备、解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。



#### 加载

类装载器把编译好的class文件载入内存, 创建相关的Class对象(封装了我们要使用的类类型信息)

##### 1.1 加载过程

1. 通过全限定类名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为方法去的运行时数据结构
3. 在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

#### 类加载器

##### 2.1 简述

实现类加载阶段中的 "通过一个类的全限定名来获取描述此类的二进制字节流" 这个动作的代码模块称为-类加载器。

将 class 文件二进制数据放入方法区内, 然后在堆内(heap)创建一个 java.lang.Class 对象, Class 对象封装了类在方法区内的数据结构, 并且向开发者提供了访问方法区内的数据结构的接口.

##### 2.2 应用领域

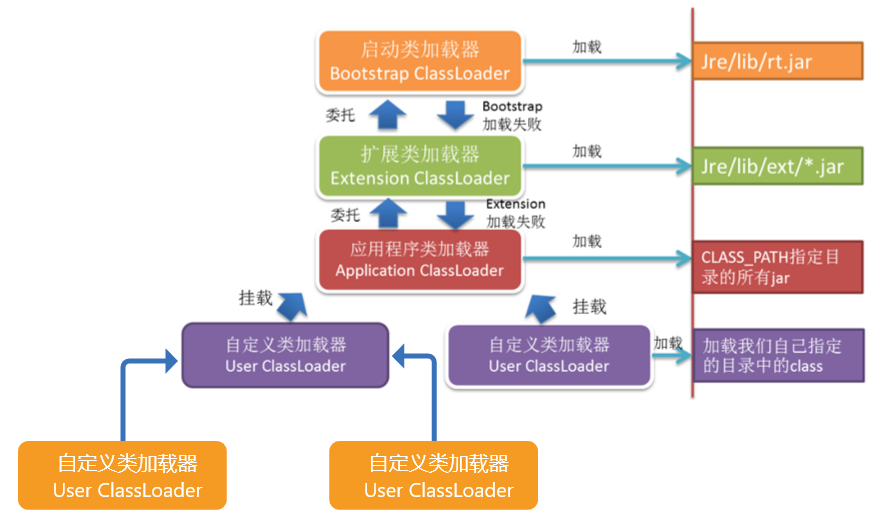
只要运行java代码, 类加载器就是必不可少的。 对于下面几个领域是非常重要的。

1. 类层次划分:
2. OSGi: 动态模型系统, 类似热部署
3. 热部署: 运行时升级软件却不需要重启
4. 代码加密

##### 2.3 双亲委派模型

当一个类加载器收到了加载请求, 它先把这个请求委派给父类加载器去完成, 每一个层次的类加载器都是如此。 当父加载器无法完成这个加载请求时, 子类加载器才会尝试自己去加载。

对于任意一个类, 都需要其本身和它的加载器一同来确认它的唯一性, 比如是同一个class文件, 它的类加载器不同, 那么这两个类也不相等。



###### 2.3.1 应用程序(系统)类加载器

Application ClassLoader: java编写, 独立于虚拟机之外, 继承抽象类java.lang.ClassLoader。 可由ClassLoader.getSystemClassLoader()返回, 默认的类加载器, 加载用户类路径ClassPath上所指定的类库。

###### 2.3.2 扩展加载器

Extension ClassLoader: java编写, 独立于虚拟机之外, 继承抽象类java.lang.ClassLoader

###### 2.3.3 启动类加载器

Bootstrap ClassLoader: C++ 实现的, 是虚拟机自身的一部分。

##### 2.4破坏双亲委派模型

#### 链接

##### 2.1 验证

确保class文件的字节流中包含的信息符合当前虚拟机的要求, 并且不会危害虚拟机自身的安全。文件格式验证、元数据验证、字节码验证、符号引用验证。可以使用-Xverify:none 来关闭大部分的类验证措施, 以缩短虚拟机加载的时间。

##### 2.2 准备

为静态变量分配内存空间, 并设置默认值, 换可能会给一些数据结构分配内存(例: 方法表)提高程序性能.

##### 2.3 解析

在类型的常量池中寻找类、接口、字段和方法的符号引用，把这些符号引用替换成直接引用。这个阶段可以被推迟到初始化之后，当程序运行的过程中真正使用某个符号引用的时候 再去解析它。

#### 初始化

<clinit>() 类在首次主动使用时执行初始化, 为类(静态)变量赋予正确的初始值. 一个正确的初始值是通过类变量初始化语句或者静态初始化块给出的

##### 初始化时机

**3.1.1 主动引用触发初始化**

1. 创建类的实例

2. 调用类的静态方法

3. 使用类的非常量静态字段

4. 调用Java API中的某些反射方法

5. 初始化时父类未初始化, 先触发父类的初始化（而一个接口在初始化时，并不要求其父接口全部都完成了初始化）

6. 含有main()方法的类启动时

7. 当使用 JDK 1.7 的动态语言支持时, 如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF\_getStatic、REF\_putStatic、REF\_invokeStatic 的方法句柄, 并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

**3.1.2 被动引用不触发初始化**

1. 通过子类引用父类的静态字段, 不会导致子类初始化。

2. 通过数组定义来引用类, 不会触发此类的初始化。MyClass[] cs = new MyClass[10];

3. 常量在编译阶段会存入调用类的常量池中, 本质上并没有直接引用到定义常量的类, 因此不会触发定义常量的类的初始化。

#### 对象的创建过程

1. 确认类元信息是否存在

类元数据metadata即java classes 在java hotspot vm 内的表示。类元数据在持久代中分配, jdk1.8之后持久代移除, 分配在本地内存中由metaspace区服。

即首先在metaspace中寻找该类元信息是否存在, 不存在的话由双亲委派模型来寻找加载, 找到的话生成(加载 -> 验证 -> 准备 -> 解析 -> 初始化)对应的Class对象, 不存在的话ClassNotFoundException

1. 分配对象内存

首先计算对象占用空间大小, 如果实例成员变量是引用变脸, 仅分配引用变量空间即4个字节大小。接着在堆中划分一块内存空间给新对象。

\* 在分配内存空间时, 需要通过CAS, 区域枷锁等方式保证分配操作的原子性。

1. 设置默认值
2. 设置对象头

设置新对象的哈希码, GC信息, 锁信息, 对象所属的类元信息等，取决于JVM的实现。

1. 执行init方法

初始化成员变量, 执行实例化代码块, 调用类的构造方法, 并把堆内对象的首地址赋值给引用变量。

#### 参考文献

1. https://www.cnblogs.com/czwbig/p/11127222.html

### 静态代理和动态代理

#### 静态代理类

由程序员创建或由特定工具自动生成源代码, 再对其编译. 在程序运行前, 代理类的.class文件就已经存在了.

静态代理只能为特定的接口提供服务, 如果想为多个接口提供服务, 则需要创建多个代理类。 即静态代理进行扩展将非常困难。

#### 动态代理

在程序运行时, 运用反射机制动态创建而成。 动态代理解决了静态代理面临的局限性。 即aop (aspect oriented programming).

### 其它

Java支持的最大线程数：

没有限制, 取决于CPU, 操作系统和java版本等多个因素. 拥有现代CPU和1-2G内存很容易支持上千个线程的JAVA虚拟机。

Java对象的大小:

通常引用的大小是4个字节, Object obj = new Object(); 通常new的Object大小为8个字节, 所以所有其他对象的大小都大于8个字节, 那么上面的最小为4(引用)+8(对象)=12个字节. 且对象分配空间是以8的整数倍进行分配的.

## 并发

### 线程安全

#### 造成线程安全的原因

1. 存在共享数据(也称临界资源: 各进程采取互斥的方式, 实现共享的资源称作临界资源)
2. 存在多个线程共通操作临界资源

### Thread

#### 状态

1. 初始(NEW): 新创建了一个线程对象. 但还没有调用start()方法.

2. 运行(RUNNABLE): Java线程中将就绪(ready)和运行中(running)两种状态笼统的成为"运行".

线程对象创建后. 其他线程(比如main线程)调用了该对象的start()方法. 该状态的线程位于可运行线程池中. 等待被线程调度选中. 获取cpu 的使用权. 此时处于就绪状态(ready). 就绪状态的线程在获得cpu 时间片后变为运行中状态(running).

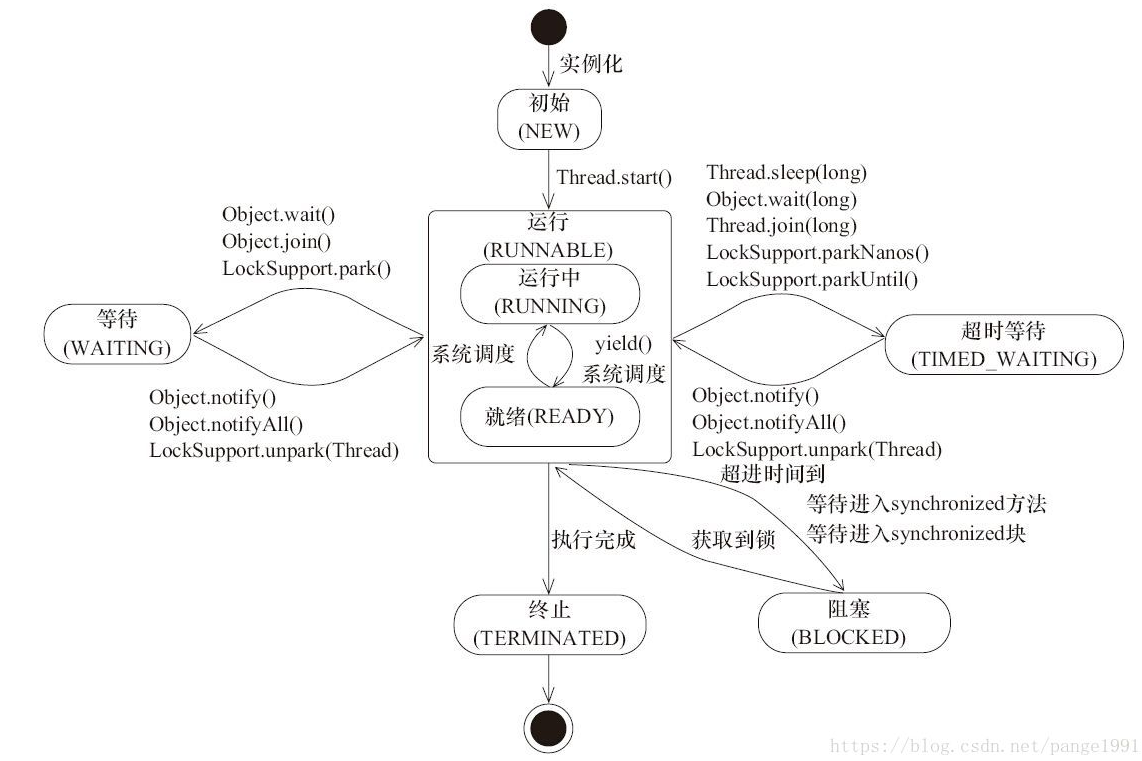
3. 阻塞(BLOCKED): 表线程阻塞于锁.

4. 等待(WAITING): 进入该状态的线程需要等待其他线程做出一些特定动作(通知或中断).

5. 超时等待(TIME\_WAITING): 该状态不同于WAITING. 它可以在指定的时间内自行返回.

6. 终止(TERMINATED): 表示该线程已经执行完毕.

附录: https://www.cnblogs.com/cowboys/p/9315331.html



#### 线程相关方法

interrupt(): Thread的实例方法, 调用后修改该线程标记为中断状态. 该线程仍会继续执行. 若使用的线程处于阻塞状态(wait sleep join), 调用后会唤起线程, 直接抛出InterruptedException异常, 并清除中断状态. 特例: 如果是因为synchronized获取锁时进入阻塞状态, 调用该方法并不能抛出异常, 只是修改状态, 它还会一直等待获取锁.

isInterrupted(): Thread的实例方法, 调用后返回该线程是否是中断状态.

interrupted(): Thread的类方法, 返回该方法所处当前环境的线程的中断状态, 之后清除中断状态.

sleep(millSecond) : thread的类方法, 线程进入Waiting状态, 不释放锁. 给其它线程执行创造了机会.

yield() : thread的类方法, 放弃获取的cpu时间片, 进入就绪状态, 不会导致阻塞.

join() : Thread的实例方法, 当前线程里调用其它线程t的join方法, 当前线程进入WAITING/TIME\_WAITING状态, 当前线程不释放已经持有的对象锁. 线程t执行完毕或者millis时间到, 当前线程进入就绪状态。

**当使用synchronized的时候锁对象会拥有关联的monitor对象, 即此时该锁对象会维护两个集合, Entry Set和Wait Set, 即锁池和等待池. {**

wait() : 拥有锁时由锁对象调用, 当前线程进入WAITING状态, 且进入到wait set中, 需要notify和notifyall唤醒. wait之后会立即放掉锁, 即释放资源.

notify() : 拥有锁时由锁对象调用, 唤醒wait set中的随机一个线程加入到Entry Set中, WAITING->RUNNABLE.

notifyAll() : 拥有锁时由锁对象调用, 唤醒wait set中的全部线程加入到Entry Set中, 全部由WAITING->RUNNABLE. **}**

### ThreadPool

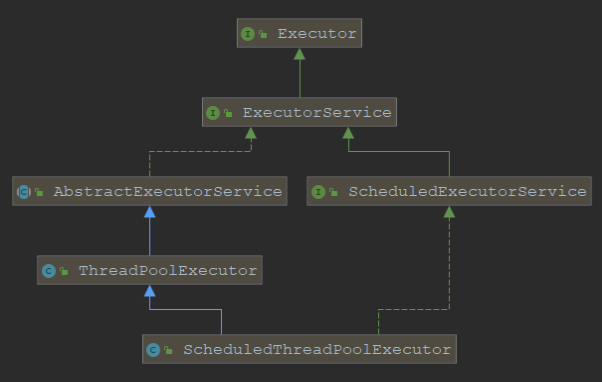
线程池, jdk1.5之后通过框架的方式来控制线程的生命周期, 并且可以避免this指针逃逸(构造函数返回之前, 其他线程已经就持有了该对象的应用, 产生的结果自然和预期可能会产生差异, 例NullPointException. 常见的this指针逃逸: 在构造函数中注册事件监听, 在构造函数中启动新线程)问题.

> 降低资源消耗：重复利用已创建的线程降低线程创建和销毁造成的消耗

> 提高响应速度：任务到达时，任务不需要等待线程创建

> 提高线程的可管理性：可以对线程统一分配、调优和监控。

#### 接口相关描述



##### Executor

一个接口, 其定义了一个接收Runnable对象的方法executor. 其方法签名为executor(Runnable command). 该方法接收一个Runable实例, 它用来执行一个任务, 任务即一个实现了Runnable接口的类.

##### ExecutorService

是一个比Executor使用更广泛的子类接口, 其提供了生命周期管理的方法, 返回 Future 对象以及可跟踪一个或多个异步任务执行状况返回Future的方法.

##### Executors

提供了一系列工厂方法用于创建线程池, 返回的线程池都实现了ExecutorService接口.

> public static ExecutorService newFiexedThreadPool(int Threads) 创建固定数目线程的线程池

> public static ExecutorService newCachedThreadPool() 创建一个可缓存的线程池, 调用execute 将重用以前构造的线程(如果线程可用). 如果没有可用的线程, 则创建一个新线程并添加到池中. 终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

> public static ExecutorService newSingleThreadExecutor()创建一个单线程化的Executor

> public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

以上的都存在缺点, 或者是任务数没有限制, 或者是线程数没有做限制.

##### ThreadPoolExecutor

|  |  |  |
| --- | --- | --- |
| **字段** | **类型** | **描述** |
| corePoolSize | int | 线程池中所保存的核心线程数, 包括空闲线程 |
| maximumPoolSize | int | 池中允许的最大线程数 |
| keepAliveTime | long | 线程池中的空闲线程所能持续的最长时间 |
| unit | TimeUnit | 持续时间的单位 |
| workQueue | BlockimgQueue<Runnable> | 任务执行前保存任务的队列, 仅保存由execute方法提交的Runnable任务 |
| threadFactory | ThreadFactory | 线程创建工厂 |
| handler | RejectedExecutionHandler | 拒绝策略 |

**corePoolSeize, maximumPoolSize**

Runtime.getRuntime().availableProcessors(); 取得当前系统的cpu核数.

对于CPU密集型(大部分时间用来做计算, 逻辑判断等动作的程序称之为CPU bound, 例计算圆周率)的任务, 若最大线程数大于cpu内核数(cpu内核数==同时最大执行线程数), 那么根据cpu的调度算法有的可能会频繁的切换而带来很大的开销。所以推荐设置为 CPU核心数 或者 CPU核心数+1

对于IO密集型(涉及到网络, 磁盘IO的任务都是IO bound[对于IO密集型, 任务越多cpu效率越高], 例web应用)一个线程对应一个方法栈, 线程的生命周期与方法栈相同, 由于IO需要耗费时间是的该线程一直再等待IO所以可以扩大线程数。2倍CPU个数+1

计算方式: https://www.cnblogs.com/warehouse/p/10810338.html

**当新提交一个任务时：**

1. 如果poolSize<corePoolSize, 新增加一个线程处理新的任务。
2. 如果poolSize=corePoolSize, 新任务会被放入阻塞队列等待。
3. 如果阻塞队列的容量达到上限, 且这时poolSize<maximumPoolSize, 新增线程来处理任务。
4. 如果阻塞队列满了, 且poolSize=maximumPoolSize, 那么线程池已经达到极限, 会根据饱和策略RejectedExecutionHandler拒绝新的任务。

#### ExecutorService

shutdown()

平滑地关闭 ExecutorService,导致ExecutorService停止接受任何新的任务且等待已经提交的任务执行完成(已经提交的任务会分两类：一类是已经在执行的，另一类是还没有开始执行的)，当所有已经提交的任务执行完毕后将会关闭ExecutorService.

submit()

方法返回的 Future 对象，可以调用isDone()方法查询Future是否已经完成。当任务完成时，它具有一个结果，你可以调用get()方法来获取该结果。你也可以不用isDone()进行检查就直接调用get()获取结果，在这种情况下，get()将阻塞，直至结果准备就绪，还可以取消任务的执行。Future 提供了 cancel() 方法用来取消执行 pending 中的任务

### CAS

Compare and Swap 比较并设置, 用于在硬件层面上提供原子性操作, 在 Intel 处理器中, 比较并交换通过指令cmpxchg实现, 比较是否和给定的数值一致, 如果一致则修改, 不一致则不修改.

do{

备份旧数据；

基于旧数据构造新数据；

}while(!CAS(内存地址, 备份的旧数据, 新数据))

在java中CAS主要由sun.misc.Unsafe这个类通过JNI调用CPU底层指令实现.

### Synchronized

#### 概要

Synchronized可以保证方法或者代码块在被访问时, 同一时刻只有一个线程对该资源进行访问, 且可以保证共享内存变量的可见性(完全可以替代Volatile功能)。 可以用在代码块上, 方法上, 类方法上. 非公平锁, 可重入锁.

> synchronized关键字不能被继承.

> 定义接口方法时, 不能使用synchronized关键字.

> 构造方法不能使用synchronized关键字, 但可以使用代码块进行同步.

> synchronized使用对象锁或者作用于方法上, 那么同一个对象使用同一把锁. 若作用于类方法上, 或者使用类对象锁, 那么该类的所有对象使用同一把锁。

#### 底层实现

开始是重量级锁即底层使用对象关联的monitor辅助锁对象(互斥量对象)来进行锁机制, 1.6后进行了优化, 引入了偏向锁, 轻量级锁, 自旋锁, 消除锁. 这些优化手段并没有使用到monitor对象.

同步代码块使用monitorenter和monitorexit指令实现, 而方法同步是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的.

#### 重点

1. 因为synchronized是可重入的所以A对象的同步方法里面可以调用它的另一个同步方法, 当然也可以调用继承的父类种的同步方法, 只是每次重入monitor中的计数器仍会继续加1.
2. Synchronized获取锁进入阻塞后不能被阻断, 只能一直去等锁, 然后执行, 等不到就继续等.
3. 由于wait notify 和notifyAll的实现和monitor对象有关, 所以这些方法必须放在synchronizzed块中.

详细: https://blog.csdn.net/javazejian/article/details/72828483

### ReentrantLock

#### 概要

是java.util.concurrent包下提供的一套互斥锁. 它提供了等待可中断, 公平锁, 非公平锁(默认), 锁绑定多个条件这些特性.

#### 接口

##### Lock

void lock() : 获取锁, 可重入, 取不到就一直阻塞, 不能被中断。

void lockInterruptibly() : 获取锁, 可重入, 优先相应中断, 中断后抛出InterruptedException异常并清除中断标记, 终止当前线程。

boolean tryLock : 拿到锁就返回true, 拿不到就false. 可重入, 可以设置等待时间, 也检测中断。

unlock() : 释放锁

Condition newCondition() : 返回当前线程的condition可多次调用.

##### ReentrantLock

实现了Lock的接口.

构造函数, ReentranLock(boolean fair) : true创建一个公平锁, false为非公平锁.

Public方法:

int GetQueueLength() 查看有多少线程等待锁

boolean hasQueuedThreads() 是否有线程等待抢锁

boolean hasQueuedThread(Thread thread) 是否有指定线程等待锁

int getHoldCount() 当前线程是否抢到锁, 0没有.

boolean isLocked() 查询此锁是否由任何线程持有

Boolean isFair() 是否为公平锁

**Condition**

Condition是Lock上的一个条件, 可用于线程间通信, 在粒度和性能上都优于Object的通信方法(wait, notify, notifyAll)

void await() : condition线程进入阻塞状态, 释放锁即资源, 允许中断抛异常, 必须在lock域中使用。

void awaitUninterruptibly() : 不允许中断, 若中断还会继续等待唤醒

Long awaitNanos(long nanosTimeout) 设置阻塞时间, 返回时间大于0表示被唤醒, 小于等于0超时, 其它等于await.

boolean awaitUntil(Date deadline) 和awaitNanos类似只是等待由参数指定的某一时刻

boolean await(long time, TimeUnit unit) 类似awaitNanos, 被唤醒true, 超时false

void signal() 唤醒指定线程.

void signalAll() 唤醒全部线程.

### Synchronized和ReentrantLock

|  |  |  |
| --- | --- | --- |
|  | **Synchronized** | **ReentrantLock** |
| 实现方式 | 依赖JVM实现 | 依赖JDK实现 |
| 使用 | 使用方便 | 手动加锁放锁 |
| 特性 | 非公平锁, 可重入 | 公平锁, 非公平锁, 可重入 |
| 粒度 | 粗粒度 | 细粒度, 更灵活 |
| 性能 | 1.6之后也很快 官方推荐 | 很快 api实现级别的 |

### 锁

根据锁的特性, 对锁进行分类.

#### 乐观锁悲观锁

体现在线程是否要同步, 数据库中对此概有念实际应用, 参照Mysql数据库章节。

**悲观锁**

对于同一数据的并发操作, 悲观锁总是认为在自己使用数据时有别的线程来修改数据. 常见的java的synchronized和lock以及数据库的事务都是采用的悲观锁. 常用于写操作.

**乐观锁**

对于同一数据的并发操作, 乐观锁认为在自己使用数据时没有别的线程来修改数据. java原子类的递增操作采用的是CAS自旋实现的, 以及数据库的MVCC也是乐观锁的一种实现.

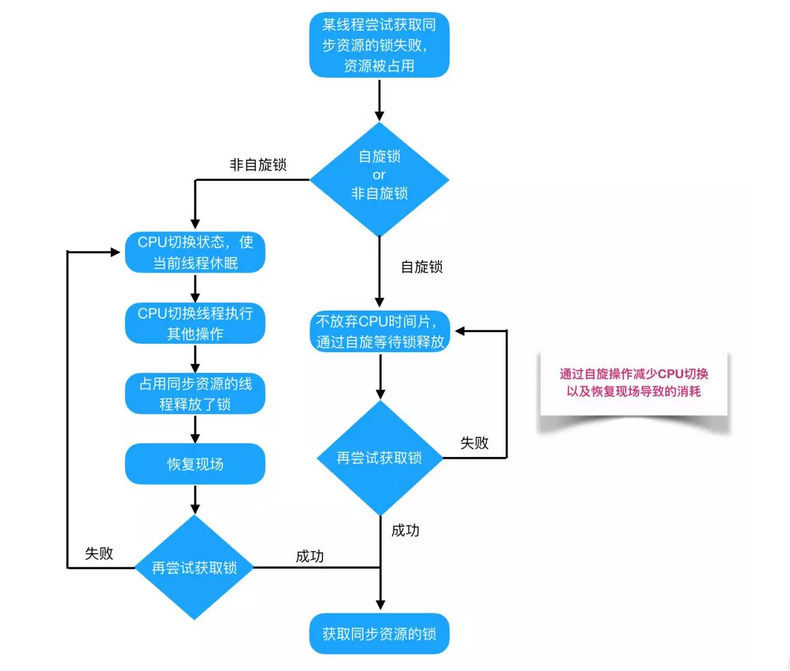
#### 自旋锁和适应性自旋锁

**自旋锁**

线程获取一个正被其它线程使用中的资源时, 会进入阻塞状态, 而进入阻塞状态需要cpu修改状态等一系列操作. 若该资源占用的时间非常短暂, 比cpu修改状态更快, 那么通过自旋无疑更好。Jdk6中默认开启, 并引入自适应自旋锁.

**自适应自旋锁**

根据上一次相同对象的锁的自旋情况来决定下次自旋是否继续。



#### 无锁, 偏向锁, 轻量级锁, 重量级锁

Synchronized在jdk1.5之前使用的就是重量级锁. 1.6之后实现下面的几种锁机制。

无锁, 不加锁, 对共享资源进行不断的循环尝试修改(类似于CAS), 直至修改成功, 适用于并发很少的情况。

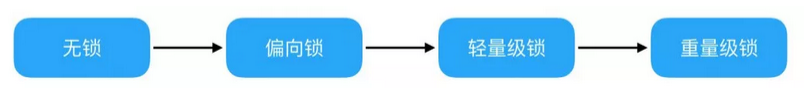
偏向锁, 在对象头和栈帧中的锁记录里存储锁偏向的线程 ID, 以后该线程在进入和退出同步块时不需要花费 CAS 操作来加锁和解锁. 适用于不存在多线程竞争，而且总是由同一线程多次获得的情况.

轻量级锁, 当多个线程通过自旋的方式获取偏向锁时, 升级为轻量级锁.

重量级锁, 当多个线程通过自选的方式获取不到锁时膨胀为重量级锁。依赖于操作系统 Mutex Lock 互斥锁实现, 比较消耗cpu.

|  |  |  |  |
| --- | --- | --- | --- |
| 锁 | 优点 | 缺点 | 适用场景 |
| 偏向锁 | 加锁和解锁不需要额外的消耗, 和执行非同步方法比仅存在纳秒级的差距 | 如果线程间存在锁竞争, 会带来额外的锁撤销的消耗 | 适用于只有一个线程访问同步块场景 |
| 轻量级锁 | 竞争的线程不会阻塞, 提高了程序的响应速度 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU | 追求响应时间  同步块执行速度非常快 |
| 重量级锁 | 线程竞争不使用自旋, 不会消耗CPU | 线程阻塞, 响应时间缓慢 | 追求吞吐量  同步块执行速度较长 |

锁只能升级不能降级.



附录: https://www.cnblogs.com/cxuanBlog/p/11684390.html

#### 公平锁和非公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁, 线程直接进入队列中排队, 队列中的第一个线程才能获得锁. 公平锁的优点是等待锁的线程不会饿死. 缺点是整体吞吐效率相对非公平锁要低, 等待队列中除第一个线程以外的所有线程都会阻塞, CPU唤醒阻塞线程的开销比非公平锁大.

非公平锁是多个线程加锁时直接尝试获取锁, 获取不到才会到等待队列的队尾等待. 但如果此时锁刚好可用, 那么这个线程可以无需阻塞直接获取到锁, 所以非公平锁有可能出现后申请锁的线程先获取锁的场景. 非公平锁的优点是可以减少唤起线程的开销, 整体的吞吐效率高, 因为线程有几率不阻塞直接获得锁, CPU不必唤醒所有线程. 缺点是处于等待队列中的线程可能会饿死, 或者等很久才会获得锁. Synchronized是非公平锁。

#### 可重入锁和非可重入锁

可重入锁(递归锁), 当一个线程获取锁没有释放后, 在接下来个的业务中还需要获取同样的一个锁时, 它可以直接获取锁, 锁标记继续+1. Java中synchronized和ReentrantLock都是可重入锁.

非可重入锁与之相反.

#### 共享锁和排它锁(独享锁)

共享锁, 该锁可以被多个线程共享持有, 若数据A加上共享锁后不能再加排他锁, 且其它线程只能加共享锁, 获取共享锁的线程只能读数据.

独享锁, 一次只能被一个线程所有, 可以进行读修改. 不能被其它线程加以其它锁.

### 共享变量的可见性

线程在工作的时候有自己的私有内存，工作内存。程序运行的时候从主内存拉取需要的变量到工作内存。

1. Volatile 修饰的变量，不论什么语句都会从主内存拉取变量
2. Synchronized 加锁的代码块， 系统会从主内存拉取变量
3. 如果调用了Synchronized 加锁的方法，或者方法中有Synchronized 那么这个方法中的变量会从主内存中取得
4. 变量被复制，或者参与运算，或者线程休眠了，那么该变量的值会从主内存拉取
5. 变量只是读了，没有参与运算，所在的方法没有Synchronized ，也没有调用有Synchronized 的方法，也没有发生线程休眠，那么该变量不会从主内存拉取。

# 数据结构扩展

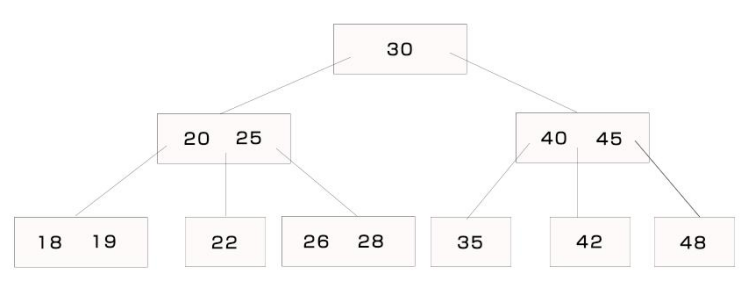
## AVL树

平衡二叉树, 层级只差不大于1, 通过旋转保证规则达到平衡性.

缺点: 树高, IO利用率不高(操作系统中数据交互以页4KB为单位交换, Mysql一页是16KB , 二叉树一个节点并占不到这莫多)

## B树 (MySql)

> 多路分支绝对平衡二叉树

> B树结构能够完美的解决磁盘的交换利用率.

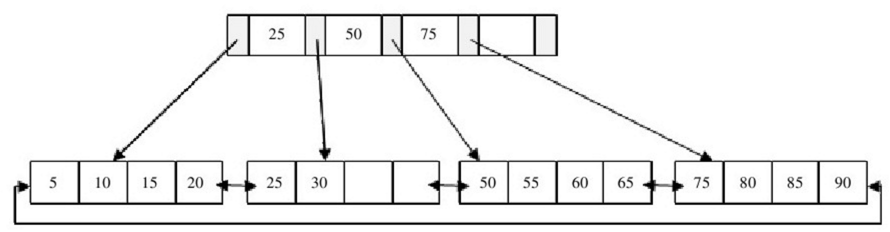
> 节点中存储多个标志, 从而有多个分支

> 数据合并和分裂来保证平衡

> 自增主键, 自增添加数据, 使得树右边调整不会整颗树进行调整

## B+ 树

> 左闭合的比较方式即 '<='

> 根节点和支节点上没有数据区, 只有叶子节点才有

> 叶子节点上的关键字形成双向的链式结构

**相比较B树的优势**

1. B+树是B树的Plus版本
2. IO效率高于B树
3. 基于索引的表扫描性能高于B-树
4. 排序能力强 范围查找
5. 基于索引的查询B+更趋于稳定

# Window

## 常用操作

### 进程

**> 查看80端口占用情况**

netstat -ano |findstr "80"

**> 查看对应进程id的应用程序**

tasklist |findstr "665120"

**> 杀死该进程**

taskkill /f /t /im 665120

# Linux

## 常用命令

### 系统命令

lsb\_release -a # 查看操作系统的版本信息

java -version # 查看java的版本信息

mvn -v # 查看maven的版本信息

cat /proc/cpuinfo | grep "physical id" | uniq | wc -l # 查看cpu个数

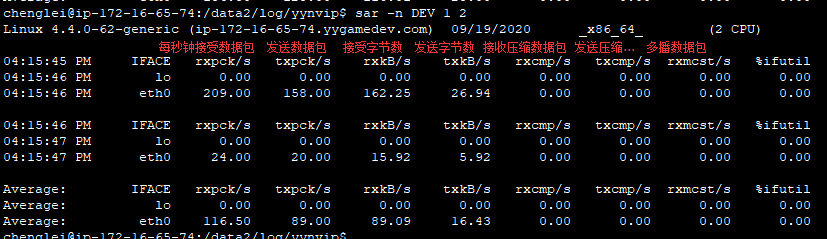
cat /proc/cpuinfo | grep "cpu cores" | uniq # 查看cpu核数

cat /proc/meminfo | grep MemTotal # 查看内存大小

### 网卡流量

#### Sar

**Sar -n DEV 1 2**



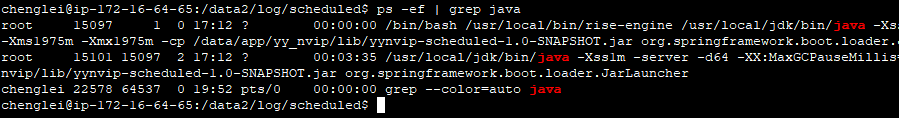
### 进程相关

#### PS

Process Status [列出系统当前运行进程的快照]

**ps -ef | grep java**

查询占用的进程号, 已经启动命令, 子进程



**ps -eo pid,ppid,%cpu,%mem,vsz,rss,comm | grep java**

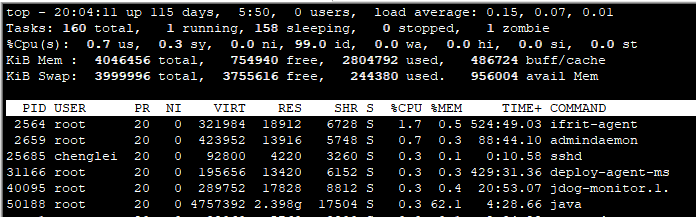
输出指定的信息, pid(进程号), ppid(子进程号), %cpu(cpu的使用率), %mem(内存占用率), vsz(占用虚拟内存大小), rss(占用固定内存大小), comm(启动命令), command(启动完整命令)



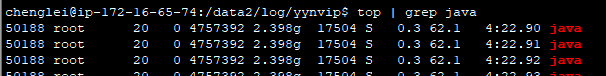
#### Top

提供实时的系统状态信息.

**top**



**top | grep java**



### 端口相关命令

netstat -tunlp # 查看端口的占用情况

### 文件操作

Chown 修改的所有者 文件名

Chgrp 所属组名 文件名

Du -h ./ 查看当前目录文件夹的大小

### 发送WebSocket请求

curl --include \

--no-buffer \

--header "Connection: Upgrade" \

--header "Upgrade: websocket" \

--header "Host: 127.0.0.1:9092" \

--header "Origin: https://echo.websocket.org" \

--header "Sec-WebSocket-Key: NVwjmQUcWCenfWu98asDmg==" \

--header "Sec-WebSocket-Version: 13" \

http://127.0.0.1:9092/socket/xxxx

### Vim

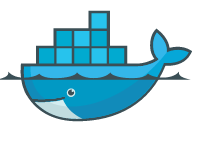
查询关键字 / 输入查询关键字 按 n | N 进行下一个

替换(全部) :%s/xxx/xxx/g

# Docker

### 介绍

#### 什么是容器

虚拟机(虚拟机代表: VMWare, OpenStack), 模拟运行一整套操作系统(包括内核, 应用运行态环境和其它系统环境)和跑在上面的应用.

容器(container), 独立运行的一个或者一组应用, 以及他们必需的运行环境。 容器是Pass(platform-as-a-service)的一种体现。 通俗的说容器是一种沙盒技术或者说是个轻量级的虚拟机。

详细: https://blog.csdn.net/gui951753/article/details/81148067

#### 什么是Docker

源于百度 - Docker 是一个开源的应用容器引擎, 让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中, 然后发布到任何流行的 Linux或Windows 机器上, 也可以实现虚拟化。容器是完全使用沙箱机制, 相互之间不会有任何接口。

通俗的讲, 容器可以看成是一个轻量级的虚拟机, Docker或是用于管理容器的一种引擎技术。 曾经的Java是提出的是"Write Once, Run Anywhere", 而现在的Docker提出的是"Build, Skip and Run Any APP, Anywhere"

优点: Docker解决了运行环境和配置问题, 方便发布, 也就方便做持续集成。更轻量的虚拟化, 节省了虚拟机的性能损耗。

缺点: 缺乏成熟的管理工具。

其它:

\* go语言实现,

\* 打出来的包叫做镜像, 镜像运行时叫做容器。Docker解决了运行环境和配置问题, 方便发布, 也就方便做持续集成。

\* Docker是基于Linux 64bit的, 在32位机器上无法运行

详细: https://baijiahao.baidu.com/s?id=1614385619899407709&wfr=spider&for=pc

docker有两个版本 社区版(Community edition, CE)和企业版(Enterprise edition, EE)。企业版包含了一些收费服务，一般我们都用社区版。

### 安装

**1. 查看linux系统版本**

Docker运行在CentOS7以上, 要求系统为64位, 系统内核版本为3.10以上。

查看linux内核版本的命令:

uname -r 或者 cat /proc/version

**2. 安装**

1) yum update // yum更新到最新

2) yum -y install docker-io

3) 安装完成查看docker版本: docker version

或者直接使用, 一键安装命令

curl -sSL https://get.daocloud.io/docker | sh

**3. 启动**

systemctl start docker

设置开机启动: systemctl enable docker

**4. 卸载docker**

yum remove docker-ce

rm -rf /var/lib/docker

### 常用命令

docker version | docker info // 产看docker的安装信息

Service docker start | systemctl start docker // docker的启动

docker images // 查看本地的镜像有哪些

docker image rm [imageName]

docker search 镜像命令 // 搜索镜像

docker pull 镜像名称 // 下载docker镜像

docker run 镜像名称 [启动容器里的执行命令] // 会根据镜像创建一个容器

docker ps [-a] // 查看当前运行的容器 -a是所有的包括停止的死亡的等

docker start|stop|restart|rm 容器ID或name

docker logs 容器ID或者name // 查看容器的标准输出

docker exec -it 容器ID或name bash // 进入容器交互界面 exit退出或者 ctrl+P ctrl+q

docker port 容器ID或name // 查看容器端口号映射到宿主机的端口号

sudo groupadd docker // 创建docker组。

sudo usermod -aG docker $USER // 将您的用户添加到该docker组。

newgrp docker // 激活对组的更改

\* image是二进制文件, 相当于镜像, 一个image通常继承另一个image加上自己的个性化设置而生成。

\* docker命令的执行需要sudo权限, 其它用户执行需要添加到docker组中

\* /etc/default/docker 文件底部添加如下, 修改为国内镜像, 然后重新启动docker

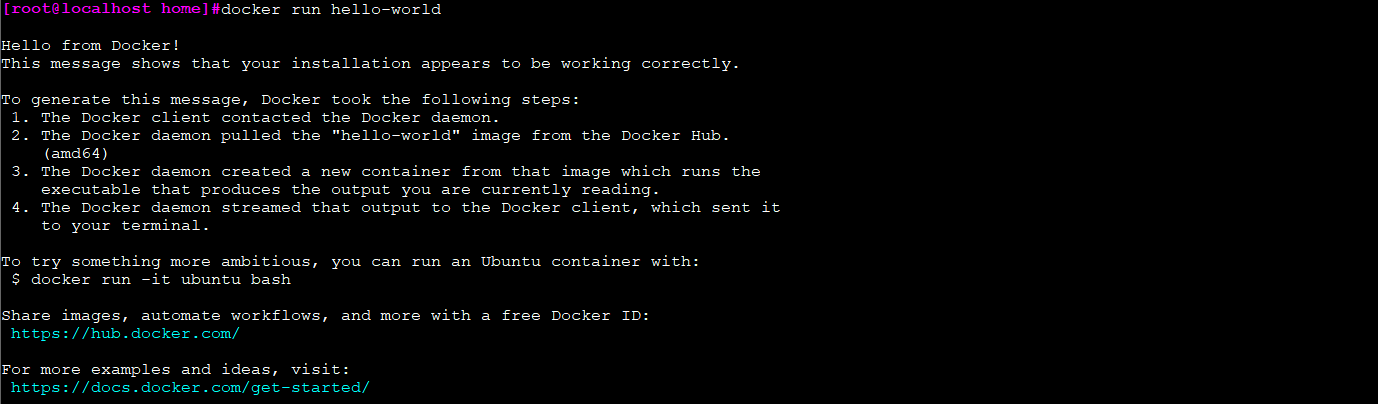
DOCKER\_OPTS="--registry-mirror=https://registry.docker-cn.com"

\* 镜像状态

Created 已创建 restarting 重启中 running|up 运行中 removing 迁移中 paused 暂停 exited 停止

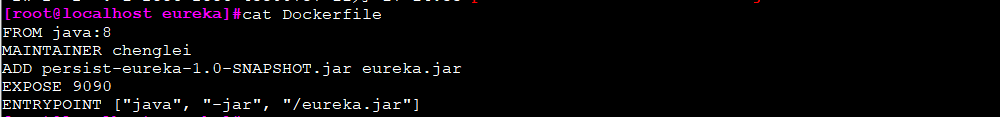
Dead 死亡

**安装成功:**

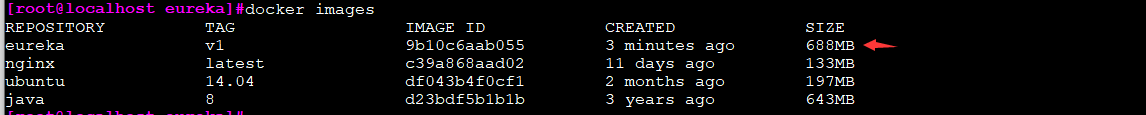


### 部署 Spring Boot

1. Spring boot 打成可执行jar
2. 将可执行jar当在linux一个目录下
3. 编写Dockerfile



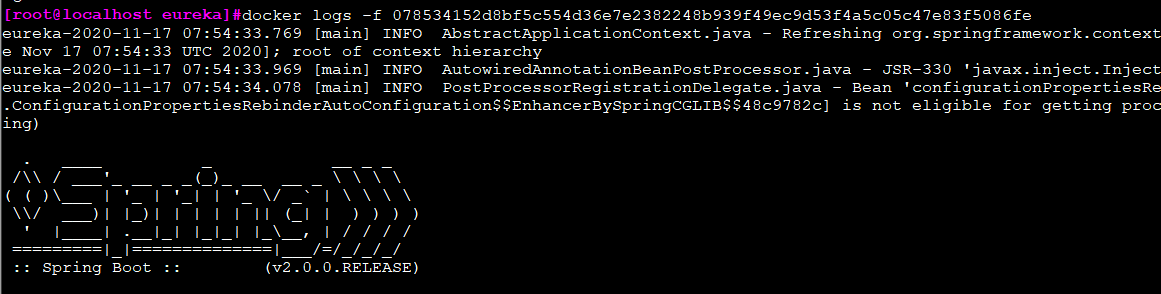
1. 执行 docker build -t eureka:v1 .
2. 查看镜像结果

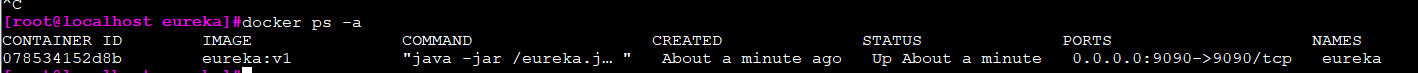


1. 执行为容器



1. 查看执行log 和状态



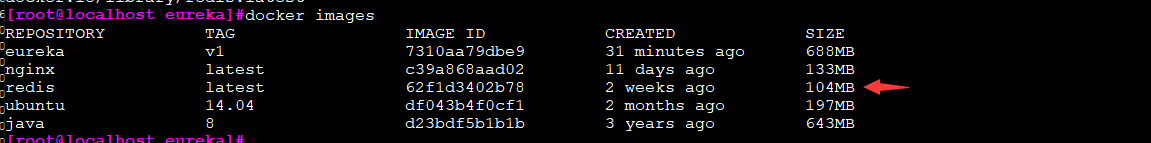


1. 访问http

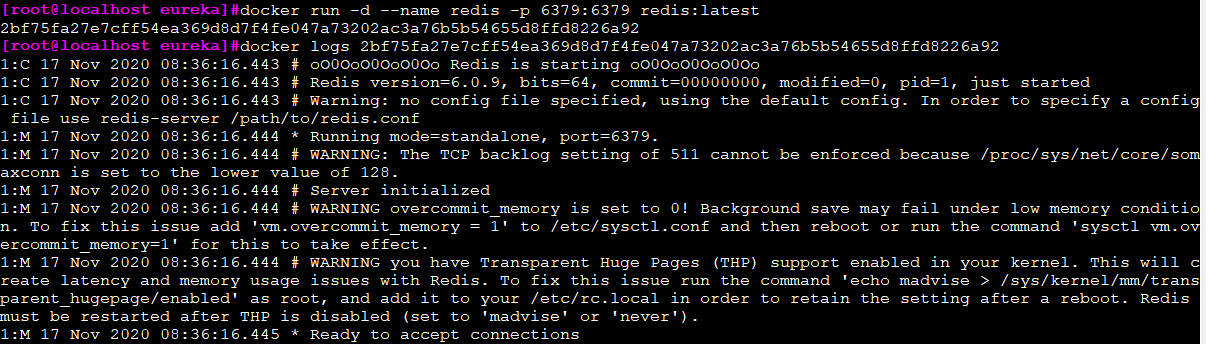


### 部署 redis

1. 拉取redis最新版本镜像 docker pull redis:latest

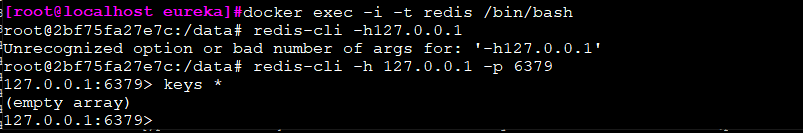


1. 运行redis容器



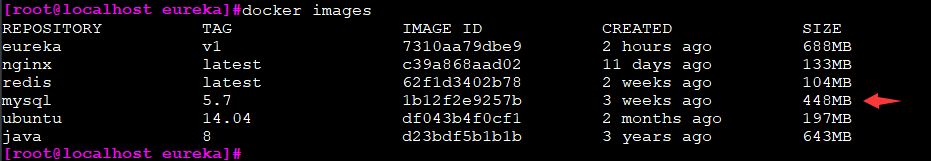


1. 使用redis

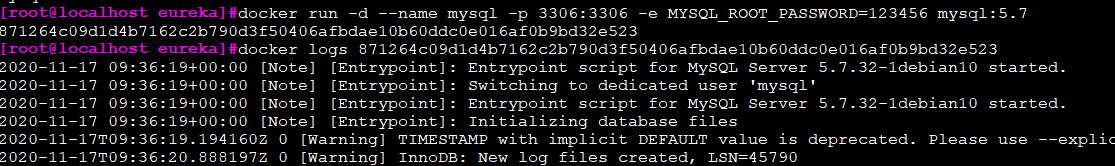


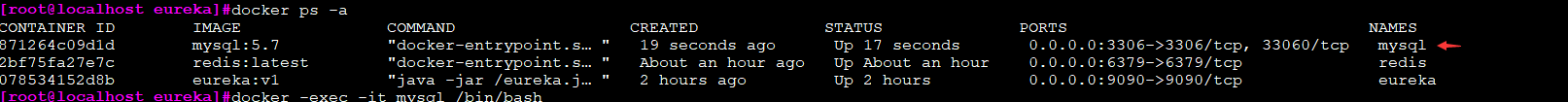
### 部署 mysql

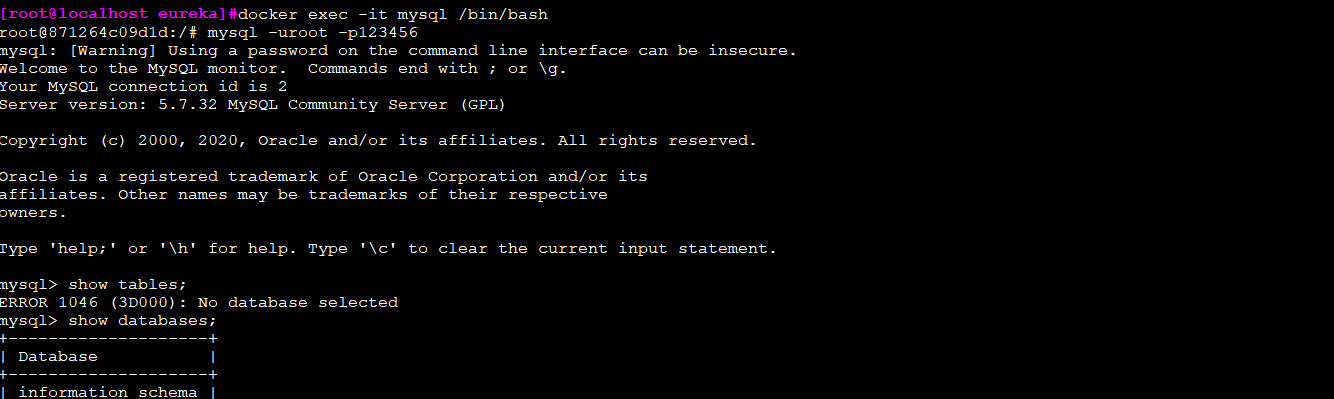
1. 拉取镜像 docker pull mysql:5.7



1. 启动容器并访问







### Compose

它是一个docker工具, 用来联系配置多个容器进行共通管理的一个工具。

# RabbitMQ

### 常用的消息中间件

ActiveMq: 很早的过时的, jms(java message service)规范, apache出品

RabbitMq: 一般, 路由分发灵活, 注重可靠性安全一致性, 性能一般, Erlang语言编写, 界面友好

RocketMq: 高并发, java开发 阿里的给了apache开源

Kafka: 超大并发, 大数据, apache 追求高吞吐, 顶级开源, 高性能, 可靠性不严格

### **作用及场景**

1. 事件驱动编程: 数据同步(全量, 增量), 数据变更进行事件通知, 客户端进行拉去来数据同步。
2. 解耦: 不同系统之间进行数据的传输, 不用强依赖。
3. 削峰: 依据队列的容量进行大流量的拦截, 后续不在进行放流.
4. 异步
5. 高内聚低耦合
6. 顺序保证
7. 事务最终一致性
8. 可靠传递

### 简介

#### AMQP

Advanced Message Queuing Protocol 应用层标准高级消息队列协议。

**Server(Broker):** 接收客户端的连接, 实现AMQP实体服务.

**Connection:** 连接, 应用程序与Server的网络连接, TCP连接.

**Channel:** 信道, 消息读写等操作在信道中进行.客户端可以建立多个信道, 每个信道代表一个会话任务.

**Message:** 消息, 应用程序和服务器之间传送的数据, 消息可以非常简单, 也可以很复杂.有Properties和Body组成.Properties为外包装, 可以对消息进行修饰, 比如消息的优先级、延迟等高级特性; Body就是消息体内容.

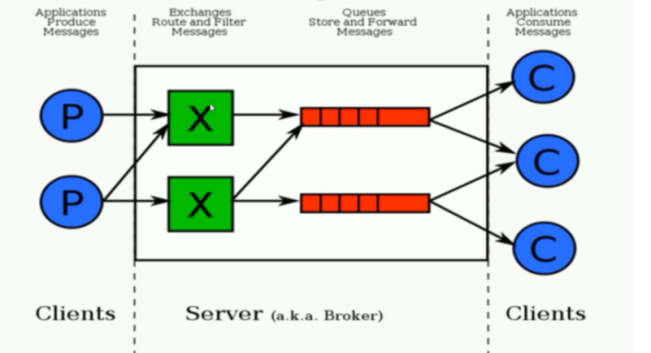
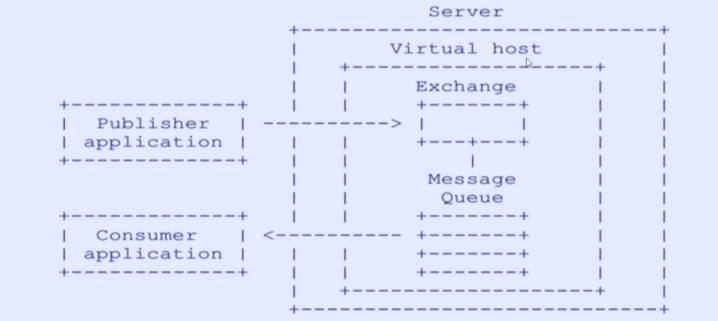
**Virtual Host:** 虚拟主机, 用于逻辑隔离.一个虚拟主机里面可以有若干个Exchange和Queue, 同一个虚拟主机里面不能有相同名称的Exchange或Queue.

**Exchange:** 交换器, 接收消息, 按照路由规则将消息路由到一个或者多个队列.如果路由不到, 或者返回给生产者, 或者直接丢弃.RabbitMQ常用的交换器常用类型有direct、topic、fanout、headers四种

**Binding:** 绑定, 交换器和消息队列之间的虚拟连接, 绑定中可以包含一个或者多个RoutingKey.

**RoutingKey:** 路由键, 生产者将消息发送给交换器的时候, 会发送一个RoutingKey, 用来指定路由规则, 这样交换器就知道把消息发送到哪个队列.路由键通常为一个"."分割的字符串, 例如"com.rabbitmq".

**Queue:** 消息队列, 用来保存消息, 供消费者消费.



#### 安装

安装和简单使用: https://blog.csdn.net/qq\_40241957/article/details/95064848

安装好的默认路径是: http://localhost:15672/

默认用户和密码: guest

Mq的web界面描述: https://www.jianshu.com/p/7b6e575fd451

Mq的学习: <https://github.com/mingyang66/spring-parent/tree/master/spring-boot-control-rabbitmq-service>

Mq的简单应用: https://blog.csdn.net/qq\_35387940/article/details/100514134

#### Web界面

#### Queue Arguments

Message ttl: 存放在队列中的存活时间

Max length: 队列的最大长度

##### QueueMessage

显示的是消息堆积后的数据.

ready是有未被消费的消息时才会有数量.

unacked是投递给消费者没有收到ACK才会有数量, 消费者接收了消息, 但是未发送ACK.

##### Ack Mode

Nack message requeue true: 获取消息不做ack应答确认, 消息重新入队

Ack message requeue false: 获取消息, 应答确认, 消息不重新入队, 将会从列表删除

Reject requeue true: 拒绝获取消息, 消息重新入队

Reject requeue false: 拒绝获取消息, 消息不重新入队, 将会被删除.

##### 常用的交换机类型

将消息转发到routing key所匹配的队列中

Direct: 直连交换机, 绝对匹配 , 即: routing key = binding key

Topic: 主题交换机, 模糊匹配, 即: com.# 可以匹配到com.a com.a.em等

# 匹配一个或多个词 \* 匹配一个词

Fanout:广播交换机, 不需要key匹配, 广播给与该交换机绑定的所有队列.

Headers: 头信息交换机, 根据消息中的header属性进行匹配, 性能差, 不推荐.

特殊的: Header Exchange, Default Exchange, 死信交换机

### 技术

#### 消息的确认

确认并保证消息被送达, 提供两种方式: 发布确认和事务(两者不可同时使用).

##### 消费者确认机制

**Auto**

> 成功消费(无异常抛出), 自动确认

> 抛出 AmqpRejectAndDontRequeueException, 消息会被决绝且requeue=false(不重入队列)

> 抛出 ImmediateAcknowledgeAmqpException 消费者会被确认

> 其它异常, 消息被拒绝 且 requeue=true, 可能发生死循环.

可以通过spring.rabbitmq.listener.simple.default-requeue-rejected: false(默认是true)去设置抛弃消息

**Manual**

手动确认, 如果没有做出应答的话, 消息不会从队列消失。

第一个参数long是deliveryTag(唯一标识 ID): 当一个消费者向 RabbitMQ 注册后, 会建立起一个 Channel, RabbitMQ 会用 basic.deliver 方法向消费者推送消息, 这个方法携带了一个 delivery tag, 它代表了 RabbitMQ 向该 Channel 投递的这条消息的唯一标识 ID, 是一个单调递增的正整数, delivery tag 的范围仅限于 Channel.

channel.basicAck(long, boolean);

确认收到消息, 消息被队列移除。 第一个参数是deliverTag信息, 第二个是true可以一次性确认传入值小于等于tag的所有消息.

channel.basicNack(long,boolean,boolean);

否认消息, 第二个参数是否批量, 第三个参数是否放回队列.

channel.basicReject(long,boolean);

拒绝消息, 第二个参数确定是否重新放入队列。

**None**

#### 死信队列

(DLX dead letter exchange)

死信:

消息被拒绝(basic.reject或basic.nack)并且requeue=false

消息TTL过期

队列达到最大长度(队列满了, 无法再添加数据到mq中)

\* 给队列(必须是主题类型的交换机)配置死信交换机才会将消息给到死信, 若不配置消息丢失.

# 云

## 概念

## 云

提供资源的网络被称为"云", 云中的资源在使用者看来是可以无限扩展的, 并且可以随时获取, 按需使用, 随时扩展, 按使用付费。通俗的说它类似于公共的水电设施的IT基础设施。

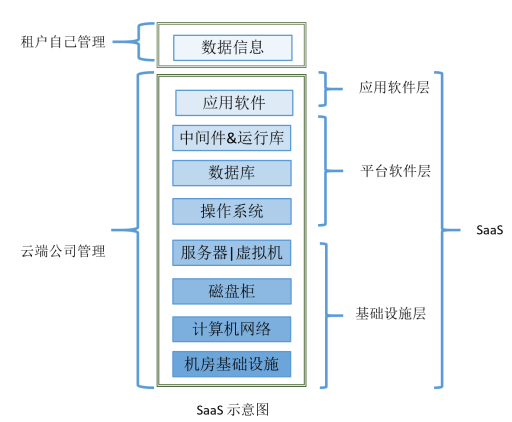
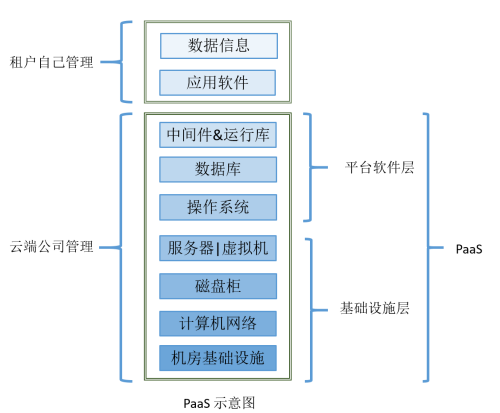
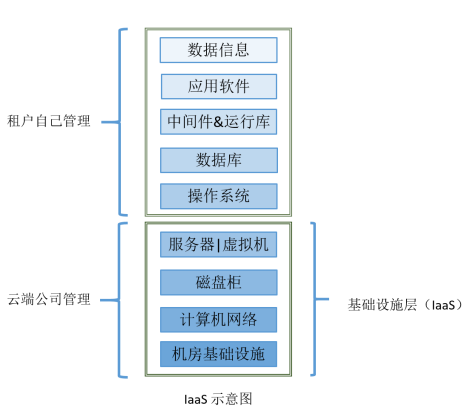
### 云计算

云计算是指IT基础设施的交付和使用模式, 通过网络以按需、易扩展的方式获得所需的服务, 这种服务可以是IT和软件(硬件、平台、软件)、互联网相关的, 也可以使任意其他的服务.

### 云服务

云服务就是指云提供的四个层面服务(基础架构即服务、平台即服务、软件即服务, 数据即服务)中的某类服务。 云服务其实是运行在云计算之上的, 算是云计算的一个具体化抽象.

### 云的一些概念



#### IMG_256IaaS

IaaS是Infrastructure as a server的缩写, 即基础设施即服务。

#### PaaS

PaaS是Platform as a Server的缩写, 意思是平台即服务, 即把运行用户所需的软件平台作为服务出租.

#### SaaS

SaaS是Software as a Server的缩写, 意思是软件即是服务.

#### DaaS

DaaS是Data as a Server的缩写, 意思数据即服务。

RBAC

基于角色的权限控制.

SCMS

SCMS是一个基于MVC的安全内容管理系统, 它的目的是从根本上抵御常见Web应用安全漏洞如：SQL注入、XSS、CSRF、Session固定/劫持等. 除了支持MySQL数据库外, 还支持PostgreSQL.

## Web Socket

# 设计模式

## 代理模式

不修改目标对象的基础上, 增强目标对象的业务逻辑。

静态代理: 事先写好的代理类, 来代替目标对象。

动态代理: 运行时, 动态生成代理对象, 代替目标对象。

JDK动态代理: 被代理类必须实现接口, 只能代理接口方法(代理类实现了相同的接口)。

CGLIB代理: 可以有也可以没有接口, 通过继承的方式代理目标对象。

# Web Service

Web Service 是用于分布式交互操作的应用程序, 为整个企业甚至多个组织之间的业务流程的集成提供了一个通用的机制。 使得不同机器上的不同应用无需借助附加的, 专门的第三方软件或者硬件, 就可相互交换数据或者集成。 具有开放性, 跨平台, 低耦合, 自包含等特性。 是构造分布式、模块化应用程序和面向服务应用集成的新技术。

### 技术

Web Service通过HTTP协议发送请求和接收结果时, 发送的请求内容和结果内容都采用XML格式封装, 并增加了一些特定的HTTP消息头, 以说明HTTP消息的内容格式, 这些特定的HTTP消息头和XML内容格式就是SOAP协议规定的。

Web service 使用xml和xsd来作为数据传递的载体, 使用soap来将它包装起来方便于不同平台, 不同软件, 不同组织之间的传递。

Soap: Simple Object Access Protocol 简单对象访问协议, 用于交换xml编码信息的轻量级协议。

其他像UDDI和WSDL技术与XML和SOAP技术紧密结合用于服务发现。

WebService服务器端首先要通过一个WSDL文件来说明自己有什么服务可以对外调用。简单的说, WSDL就像是一个说明书, 用于描述WebService及其方法、参数和返回值。 WSDL文件保存在Web服务器上, 通过一个url地址就可以访问到它。客户端要调用一个WebService服务之前, 要知道该服务的WSDL文件的地址。WebService服务提供商可以通过两种方式来暴露它的WSDL文件地址：1.注册到UDDI服务器，以便被人查找；2.直接告诉给客户端调用者。

**Web service的交互过程就是, web service遵循soap协议通过xml封装数据, 然后由http协议来传输数据。**

#### 通信

RPC远程调用和消息传递

#### 其它

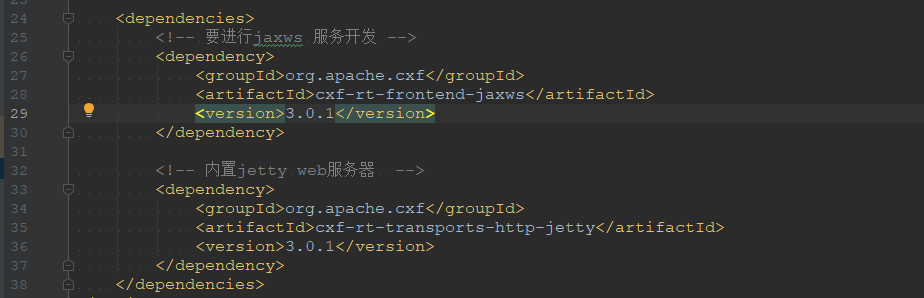
服务间常用的通信有 socket, http, web service

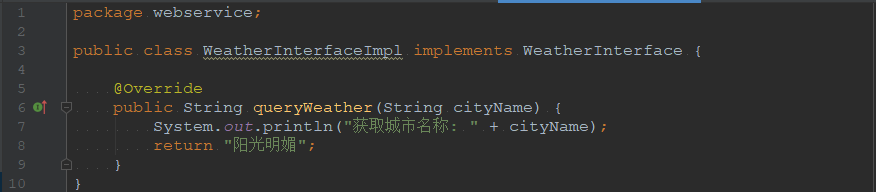
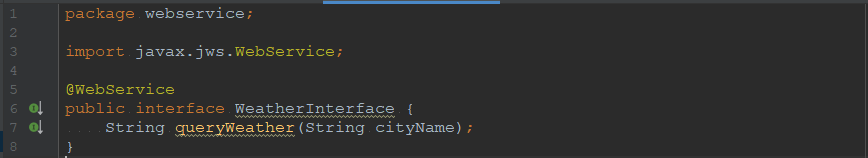
http是对socket的包装

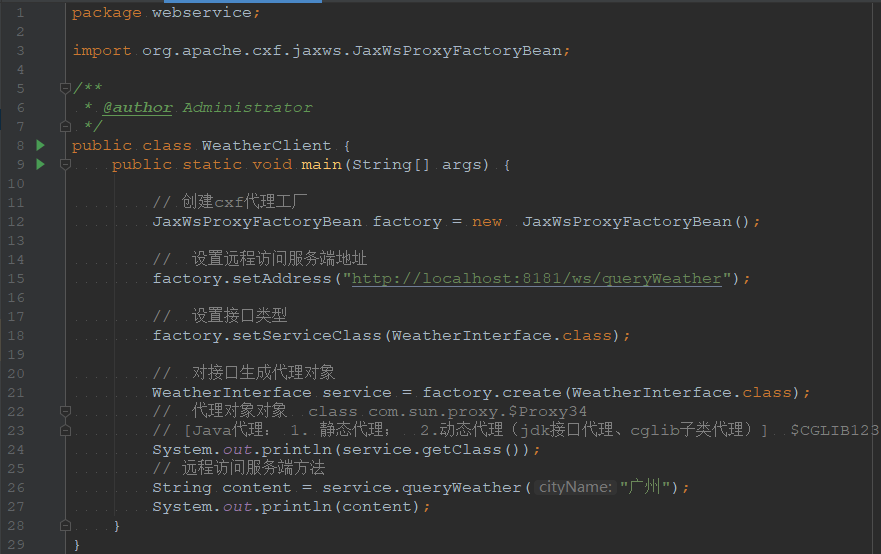
而 web service = soap(http + xml)

所以 socket < http < web service (更底层)

## 代码实现







# 其它

## IO

### 随机IO

随机IO是操作的访问地址不连续,

### 顺序IO

顺序IO是操作的访问地址连续, 产生随机IO的业务有OLTP服务, SQL, 及时消息服务等。

## 硬盘

### SSD

固态硬盘, 固态硬盘的数据读取速度非常的快, 固态硬盘的外观种类可分为四种, 分别为2.5英寸的SATA类型、M2类型的2280、M2类型的2240和M2类型的2242, 每个类型的固态硬盘都自带不同的协议, 有NVME协议、NGFF协议、PCIE协议和SATA协议等, 目前市面上性能最好的固态硬盘就是NVME协议的固态硬盘。但是要注意的是：NVME协议SSD是需要主板支持的, 在购买NVME协议的固态硬盘时必须要弄清楚电脑主板是否支持NVME协议固态硬盘。

### HDD

机械硬盘, 机械硬盘是传统的盘片+磁头的工作原理，数据都是存储在盘片上面, 磁头在盘片上面 "做运动" 来读取和处理数据信息, 相对SSD固态硬盘来说, HDD机械硬盘的速度已经完全跟不上技术革新的节奏了. 现在的机械硬盘在电脑内的作用基本上只有单纯的储存资料文件数据了。

### SSHD

混合硬盘, = SSD + HDD

\* SSD 存储在芯片上, 电子信号存储, HDD 是物理存储, 电磁转换, 通过磁头来读写肯定慢

\* SSD 速度更快, 数据更安全(HDD可能机械故障), 更快的机械请求, 更低功耗, 价格高, 存储量小

\* SSD 温度高电子活跃, 数据容易损失, 断电时间长, 电子可能会衰弱, 数据丢失

### IOPS

IOPS（Input/OutputOperations Per Second）是一个用于计算机存储设备（如硬盘（HDD）、固态硬盘（SSD）或存储区域网络（SAN））性能测试的量测方式，可以视为是每秒的读写次数。

## CAP原则

一致性(Consistency)、可用性(Availability)、分区容错性(Partition tolerance)

## 性能指标

|  |  |  |  |
| --- | --- | --- | --- |
| **指标** | **计算** | **描述** | **其它** |
| **吞吐量** | | | |
| QPS | Req/sec | 每秒请求数) |  |
| TPS |  | 每秒内的事务数 |  |
| **点击数** | | | |
| PV |  | 页面被浏览器访问次数 |  |
| UV |  | 对PV进行去重, 一个用户算一个 |  |
| **资源使用率** | | | |
| Cpu使用率 |  |  | Htop |
| 内存使用率 |  |  |  |
| 磁盘 |  |  |  |
| 网络 |  |  |  |
| **并发用户数** |  | | |
| **响应时间** | 一个服务的响应时间 | | |
| **其它** | | | |
| 错误率 |  |  |  |
| Gc率和暂停时间 |  |  |  |
| 业务指标 |  |  |  |
| 正常运行时间和服务运行状态 | |  |  |
| 日志大小 |  |  |  |

压测原则:

每天80%的访问量集中在20%的时间里，这20%的时间就叫做峰值

公式: ( 总PV数80% ) / ( 每天的秒数20% ) = 峰值时间每秒钟请求数(QPS)

### 压测小工具

**http\_load**

Web Api 压测工具

安装使用: <http://www.bjhee.com/http-load.html>

**其它压测小工具**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **实现语言** | **UI界面** | **优势** |
| Ab(ApacheBench) | c | 无 | Apache自带web压力工具, 容易上手 |
| locust | python | 有 | 支持分布式, 结果数据可以导出 |
| jmeter | java | 有 | 插件丰富, 支持生成html报告 |
| Go-stress-testing | golang | 无 | 开源, 使用简单, 无依赖, 支持websocket压测 |
| 云压测 |  | 无 | 支持更高的压测力度 |

小工具介绍: https://www.imooc.com/article/291715

### 性能检测工具

**AppDynamics**

服务性能监控/管理工具, 1> 终端用户性能体验监护 2> 计算资源监控

**New Relic**

服务器性能监控工具

**Ruxit**

## MyBatis

#开启Mybatis下划线命名转驼峰命名

mybatis.configuration.map-underscore-to-camel-case=true

## 幂等

针对一个操作不管做多少次, 产生的效果或返回的结果都是一样的。

技术方案

查询操作(Select 天然的幂等操作), 删除操作, 防止新增脏数据, 锁(悲观锁, 乐观锁, 分布式锁), 状态机幂等。